```
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_ARRAY_H
#define SPL_ARRAY_H

#include "php.h"
#include "php_spl.h"
#include "spl_iterators.h"

extern PHPAPI zend_class_entry *spl_ce_ArrayObject;
extern PHPAPI zend_class_entry *spl_ce_ArrayIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveArrayIterator;

PHP_MINIT_FUNCTION(spl_array);

extern void spl_array_iterator_append(zval *object, zval *append_value);
extern void spl_array_iterator_key(zval *object, zval *return_value);

#endif /* SPL_ARRAY_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Etienne Kneuss <colder@php.net>                             |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "zend_exceptions.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_iterators.h"
#include "spl_heap.h"
#include "spl_exceptions.h"

#define PTR_HEAP_BLOCK_SIZE 64

#define SPL_HEAP_CORRUPTED       0x00000001

#define SPL_PQUEUE_EXTR_MASK     0x00000003
#define SPL_PQUEUE_EXTR_BOTH     0x00000003
#define SPL_PQUEUE_EXTR_DATA     0x00000001
#define SPL_PQUEUE_EXTR_PRIORITY 0x00000002

zend_object_handlers spl_handler_SplHeap;
zend_object_handlers spl_handler_SplPriorityQueue;

PHPAPI zend_class_entry  *spl_ce_SplHeap;
PHPAPI zend_class_entry  *spl_ce_SplMaxHeap;
PHPAPI zend_class_entry  *spl_ce_SplMinHeap;
PHPAPI zend_class_entry  *spl_ce_SplPriorityQueue;

typedef void (*spl_ptr_heap_dtor_func)(zval *);
typedef void (*spl_ptr_heap_ctor_func)(zval *);
typedef int  (*spl_ptr_heap_cmp_func)(zval *, zval *, zval *);

typedef struct _spl_ptr_heap {
	zval                *elements;
	spl_ptr_heap_ctor_func  ctor;
	spl_ptr_heap_dtor_func  dtor;
	spl_ptr_heap_cmp_func   cmp;
	int                     count;
	int                     max_size;
	int                     flags;
} spl_ptr_heap;

typedef struct _spl_heap_object spl_heap_object;
typedef struct _spl_heap_it spl_heap_it;

struct _spl_heap_object {
	spl_ptr_heap       *heap;
	int                 flags;
	zend_class_entry   *ce_get_iterator;
	zend_function      *fptr_cmp;
	zend_function      *fptr_count;
	zend_object         std;
};

/* define an overloaded iterator structure */
struct _spl_heap_it {
	zend_user_iterator  intern;
	int                 flags;
};

static inline spl_heap_object *spl_heap_from_obj(zend_object *obj) /* {{{ */ {
	return (spl_heap_object*)((char*)(obj) - XtOffsetOf(spl_heap_object, std));
}
/* }}} */

#define Z_SPLHEAP_P(zv)  spl_heap_from_obj(Z_OBJ_P((zv)))

static void spl_ptr_heap_zval_dtor(zval *elem) { /* {{{ */
	if (!Z_ISUNDEF_P(elem)) {
		zval_ptr_dtor(elem);
	}
}
/* }}} */

static void spl_ptr_heap_zval_ctor(zval *elem) { /* {{{ */
	Z_TRY_ADDREF_P(elem);
}
/* }}} */

static int spl_ptr_heap_cmp_cb_helper(zval *object, spl_heap_object *heap_object, zval *a, zval *b, zend_long *result) { /* {{{ */
	zval zresult;

	zend_call_method_with_2_params(object, heap_object->std.ce, &heap_object->fptr_cmp, "compare", &zresult, a, b);

	if (EG(exception)) {
		return FAILURE;
	}

	*result = zval_get_long(&zresult);
	zval_ptr_dtor(&zresult);

	return SUCCESS;
}
/* }}} */

static zval *spl_pqueue_extract_helper(zval *value, int flags) /* {{{ */
{
	if ((flags & SPL_PQUEUE_EXTR_BOTH) == SPL_PQUEUE_EXTR_BOTH) {
		return value;
	} else if ((flags & SPL_PQUEUE_EXTR_BOTH) > 0) {
		if ((flags & SPL_PQUEUE_EXTR_DATA) == SPL_PQUEUE_EXTR_DATA) {
			zval *data;
			if ((data = zend_hash_str_find(Z_ARRVAL_P(value), "data", sizeof("data") - 1)) != NULL) {
				return data;
			}
		} else {
			zval *priority;
			if ((priority = zend_hash_str_find(Z_ARRVAL_P(value), "priority", sizeof("priority") - 1)) != NULL) {
				return priority;
			}
		}
	}

	return NULL;
}
/* }}} */

static int spl_ptr_heap_zval_max_cmp(zval *a, zval *b, zval *object) { /* {{{ */
	zval result;

	if (EG(exception)) {
		return 0;
	}

	if (object) {
		spl_heap_object *heap_object = Z_SPLHEAP_P(object);
		if (heap_object->fptr_cmp) {
			zend_long lval = 0;
			if (spl_ptr_heap_cmp_cb_helper(object, heap_object, a, b, &lval) == FAILURE) {
				/* exception or call failure */
				return 0;
			}
			return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
		}
	}

	compare_function(&result, a, b);
	return (int)Z_LVAL(result);
}
/* }}} */

static int spl_ptr_heap_zval_min_cmp(zval *a, zval *b, zval *object) { /* {{{ */
	zval result;

	if (EG(exception)) {
		return 0;
	}

	if (object) {
		spl_heap_object *heap_object = Z_SPLHEAP_P(object);
		if (heap_object->fptr_cmp) {
			zend_long lval = 0;
			if (spl_ptr_heap_cmp_cb_helper(object, heap_object, a, b, &lval) == FAILURE) {
				/* exception or call failure */
				return 0;
			}
			return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
		}
	}

	compare_function(&result, b, a);
	return (int)Z_LVAL(result);
}
/* }}} */

static int spl_ptr_pqueue_zval_cmp(zval *a, zval *b, zval *object) { /* {{{ */
	zval result;
	zval *a_priority_p = spl_pqueue_extract_helper(a, SPL_PQUEUE_EXTR_PRIORITY);
	zval *b_priority_p = spl_pqueue_extract_helper(b, SPL_PQUEUE_EXTR_PRIORITY);

	if ((!a_priority_p) || (!b_priority_p)) {
		zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
		return 0;
	}

	if (EG(exception)) {
		return 0;
	}

	if (object) {
		spl_heap_object *heap_object = Z_SPLHEAP_P(object);
		if (heap_object->fptr_cmp) {
			zend_long lval = 0;
			if (spl_ptr_heap_cmp_cb_helper((zval *)object, heap_object, a_priority_p, b_priority_p, &lval) == FAILURE) {
				/* exception or call failure */
				return 0;
			}
			return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
		}
	}

	compare_function(&result, a_priority_p, b_priority_p);
	return (int)Z_LVAL(result);
}
/* }}} */

static spl_ptr_heap *spl_ptr_heap_init(spl_ptr_heap_cmp_func cmp, spl_ptr_heap_ctor_func ctor, spl_ptr_heap_dtor_func dtor) /* {{{ */
{
	spl_ptr_heap *heap = emalloc(sizeof(spl_ptr_heap));

	heap->dtor     = dtor;
	heap->ctor     = ctor;
	heap->cmp      = cmp;
	heap->elements = ecalloc(PTR_HEAP_BLOCK_SIZE, sizeof(zval));
	heap->max_size = PTR_HEAP_BLOCK_SIZE;
	heap->count    = 0;
	heap->flags    = 0;

	return heap;
}
/* }}} */

static void spl_ptr_heap_insert(spl_ptr_heap *heap, zval *elem, void *cmp_userdata) { /* {{{ */
	int i;

	if (heap->count+1 > heap->max_size) {
		/* we need to allocate more memory */
		heap->elements  = erealloc(heap->elements, heap->max_size * 2 * sizeof(zval));
		memset(heap->elements + heap->max_size, 0, heap->max_size * sizeof(zval));
		heap->max_size *= 2;
	}

	/* sifting up */
	for (i = heap->count; i > 0 && heap->cmp(&heap->elements[(i-1)/2], elem, cmp_userdata) < 0; i = (i-1)/2) {
		heap->elements[i] = heap->elements[(i-1)/2];
	}
	heap->count++;

	if (EG(exception)) {
		/* exception thrown during comparison */
		heap->flags |= SPL_HEAP_CORRUPTED;
	}

	ZVAL_COPY_VALUE(&heap->elements[i], elem);
}
/* }}} */

static zval *spl_ptr_heap_top(spl_ptr_heap *heap) { /* {{{ */
	if (heap->count == 0) {
		return NULL;
	}

	return Z_ISUNDEF(heap->elements[0])? NULL : &heap->elements[0];
}
/* }}} */

static void spl_ptr_heap_delete_top(spl_ptr_heap *heap, zval *elem, void *cmp_userdata) { /* {{{ */
	int i, j;
	const int limit = (heap->count-1)/2;
	zval *bottom;

	if (heap->count == 0) {
		ZVAL_UNDEF(elem);
		return;
	}

	ZVAL_COPY_VALUE(elem, &heap->elements[0]);
	bottom = &heap->elements[--heap->count];

	for (i = 0; i < limit; i = j) {
		/* Find smaller child */
		j = i * 2 + 1;
		if(j != heap->count && heap->cmp(&heap->elements[j+1], &heap->elements[j], cmp_userdata) > 0) {
			j++; /* next child is bigger */
		}

		/* swap elements between two levels */
		if(heap->cmp(bottom, &heap->elements[j], cmp_userdata) < 0) {
			heap->elements[i] = heap->elements[j];
		} else {
			break;
		}
	}

	if (EG(exception)) {
		/* exception thrown during comparison */
		heap->flags |= SPL_HEAP_CORRUPTED;
	}

	ZVAL_COPY_VALUE(&heap->elements[i], bottom);
}
/* }}} */

static spl_ptr_heap *spl_ptr_heap_clone(spl_ptr_heap *from) { /* {{{ */
	int i;

	spl_ptr_heap *heap = emalloc(sizeof(spl_ptr_heap));

	heap->dtor     = from->dtor;
	heap->ctor     = from->ctor;
	heap->cmp      = from->cmp;
	heap->max_size = from->max_size;
	heap->count    = from->count;
	heap->flags    = from->flags;

	heap->elements = safe_emalloc(sizeof(zval), from->max_size, 0);
	memcpy(heap->elements, from->elements, sizeof(zval)*from->max_size);

	for (i=0; i < heap->count; ++i) {
		heap->ctor(&heap->elements[i]);
	}

	return heap;
}
/* }}} */

static void spl_ptr_heap_destroy(spl_ptr_heap *heap) { /* {{{ */
	int i;

	for (i=0; i < heap->count; ++i) {
		heap->dtor(&heap->elements[i]);
	}

	efree(heap->elements);
	efree(heap);
}
/* }}} */

static int spl_ptr_heap_count(spl_ptr_heap *heap) { /* {{{ */
	return heap->count;
}
/* }}} */

zend_object_iterator *spl_heap_get_iterator(zend_class_entry *ce, zval *object, int by_ref);

static void spl_heap_object_free_storage(zend_object *object) /* {{{ */
{
	spl_heap_object *intern = spl_heap_from_obj(object);

	zend_object_std_dtor(&intern->std);

	spl_ptr_heap_destroy(intern->heap);
}
/* }}} */

static zend_object *spl_heap_object_new_ex(zend_class_entry *class_type, zval *orig, int clone_orig) /* {{{ */
{
	spl_heap_object   *intern;
	zend_class_entry  *parent = class_type;
	int                inherited = 0;

	intern = zend_object_alloc(sizeof(spl_heap_object), parent);

	zend_object_std_init(&intern->std, class_type);
	object_properties_init(&intern->std, class_type);

	intern->flags      = 0;
	intern->fptr_cmp   = NULL;

	if (orig) {
		spl_heap_object *other = Z_SPLHEAP_P(orig);
		intern->ce_get_iterator = other->ce_get_iterator;
```

```
377:
378:     if (clone_orig) {
379:       intern->heap = spl_ptr_heap_clone(other->heap);
380:     } else {
381:       intern->heap = other->heap;
382:     }
383:
384:     intern->flags = other->flags;
385:   } else {
386:     intern->heap = spl_ptr_heap_init(spl_ptr_heap_zval_max_cmp, spl_ptr_heap_zval_ctor, spl_ptr_heap_zval_dtor);
387:   }
388:
389:   intern->std.handlers = &spl_handler_SplHeap;
390:
391:   while (parent) {
392:     if (parent == spl_ce_SplPriorityQueue) {
393:       intern->heap->cmp = spl_ptr_pqueue_zval_cmp;
394:       intern->flags = SPL_PQUEUE_EXTR_DATA;
395:       intern->std.handlers = &spl_handler_SplPriorityQueue;
396:       break;
397:     }
398:
399:     if (parent == spl_ce_SplMinHeap) {
400:       intern->heap->cmp = spl_ptr_heap_zval_min_cmp;
401:       break;
402:     }
403:
404:     if (parent == spl_ce_SplMaxHeap) {
405:       intern->heap->cmp = spl_ptr_heap_zval_max_cmp;
406:       break;
407:     }
408:
409:     if (parent == spl_ce_SplHeap) {
410:       break;
411:     }
412:
413:     parent = parent->parent;
414:     inherited = 1;
415:   }
416:
417:   if (!parent) { /* this must never happen */
418:     php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error, Class is not child of SplHeap");
419:   }
420:
421:   if (inherited) {
422:     intern->fptr_cmp = zend_hash_str_find_ptr(&class_type->function_table, "compare", sizeof("compare") - 1);
423:     if (intern->fptr_cmp->common.scope == parent) {
424:       intern->fptr_cmp = NULL;
425:     }
426:     intern->fptr_count = zend_hash_str_find_ptr(&class_type->function_table, "count", sizeof("count") - 1);
427:     if (intern->fptr_count->common.scope == parent) {
428:       intern->fptr_count = NULL;
429:     }
430:   }
431:
432:   return &intern->std;
433: }
434: /* }}} */
435:
436: static zend_object *spl_heap_object_new(zend_class_entry *class_type) /* {{{ */
437: {
438:   return spl_heap_object_new_ex(class_type, NULL, 0);
439: }
440: /* }}} */
441:
442: static zend_object *spl_heap_object_clone(zval *zobject) /* {{{ */
443: {
444:   zend_object        *old_object;
445:   zend_object        *new_object;
446:
447:   old_object  = Z_OBJ_P(zobject);
448:   new_object = spl_heap_object_new_ex(old_object->ce, zobject, 1);
449:
450:   zend_objects_clone_members(new_object, old_object);
451:
452:   return new_object;
453: }
454: /* }}} */
455:
456: static int spl_heap_object_count_elements(zval *object, zend_long *count) /* {{{ */
457: {
458:   spl_heap_object *intern = Z_SPLHEAP_P(object);
459:
460:   if (intern->fptr_count) {
461:     zval rv;
462:     zend_call_method_with_0_params(object, intern->std.ce, &intern->fptr_count, "count", &rv);
463:     if (!Z_ISUNDEF(rv)) {
464:       *count = zval_get_long(&rv);
465:       zval_ptr_dtor(&rv);
466:       return SUCCESS;
467:     }
468:     *count = 0;
469:     return FAILURE;
470:   }
471:
472:   *count = spl_ptr_heap_count(intern->heap);
473:
474:   return SUCCESS;
475: }
476: /* }}} */
477:
478: static HashTable* spl_heap_object_get_debug_info_helper(zend_class_entry *ce, zval *obj, int *is_temp) { /* {{{ */
479:   spl_heap_object *intern = Z_SPLHEAP_P(obj);
480:   zval tmp, heap_array;
481:   zend_string *pnstr;
482:   HashTable *debug_info;
483:   int  i;
484:
485:   *is_temp = 1;
486:
487:   if (!intern->std.properties) {
488:     rebuild_object_properties(&intern->std);
489:   }
490:
491:   debug_info = zend_new_array(zend_hash_num_elements(intern->std.properties) + 1);
492:   zend_hash_copy(debug_info, intern->std.properties, (copy_ctor_func_t) zval_add_ref);
493:
494:   pnstr = spl_gen_private_prop_name(ce, "flags", sizeof("flags")-1);
495:   ZVAL_LONG(&tmp, intern->flags);
496:   zend_hash_update(debug_info, pnstr, &tmp);
497:   zend_string_release(pnstr);
498:
499:   pnstr = spl_gen_private_prop_name(ce, "isCorrupted", sizeof("isCorrupted")-1);
500:   ZVAL_BOOL(&tmp, intern->heap->flags&SPL_HEAP_CORRUPTED);
501:   zend_hash_update(debug_info, pnstr, &tmp);
502:   zend_string_release(pnstr);
503:
504:   array_init(&heap_array);
505:
506:   for (i = 0; i < intern->heap->count; ++i) {
507:     add_index_zval(&heap_array, i, &intern->heap->elements[i]);
508:     if (Z_REFCOUNTED(intern->heap->elements[i])) {
509:       Z_ADDREF(intern->heap->elements[i]);
510:     }
511:   }
512:
513:   pnstr = spl_gen_private_prop_name(ce, "heap", sizeof("heap")-1);
514:   zend_hash_update(debug_info, pnstr, &heap_array);
515:   zend_string_release(pnstr);
516:
517:   return debug_info;
518: }
519: /* }}} */
520:
521: static HashTable *spl_heap_object_get_gc(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
522: {
523:   spl_heap_object *intern = Z_SPLHEAP_P(obj);
524:   *gc_data = intern->heap->elements;
525:   *gc_data_count = intern->heap->count;
526:
527:   return std_object_handlers.get_properties(obj);
528: }
529: /* }}} */
530:
531: static HashTable* spl_heap_object_get_debug_info(zval *obj, int *is_temp) /* {{{ */
532: {
533:   return spl_heap_object_get_debug_info_helper(spl_ce_SplHeap, obj, is_temp);
534: }
535: /* }}} */
536:
537: static HashTable* spl_pqueue_object_get_debug_info(zval *obj, int *is_temp) /* {{{ */
538: {
539:   return spl_heap_object_get_debug_info_helper(spl_ce_SplPriorityQueue, obj, is_temp);
540: }
541: /* }}} */
542:
543: /* {{{ proto int SplHeap::count()
544:  Return the number of elements in the heap. */
545: SPL_METHOD(SplHeap, count)
546: {
547:   zend_long count;
548:   spl_heap_object *intern = Z_SPLHEAP_P(getThis());
549:
550:   if (zend_parse_parameters_none() == FAILURE) {
551:     return;
552:   }
553:
554:   count = spl_ptr_heap_count(intern->heap);
555:   RETURN_LONG(count);
556: }
557: /* }}} */
558:
559: /* {{{ proto int SplHeap::isEmpty()
560:  Return true if the heap is empty. */
561: SPL_METHOD(SplHeap, isEmpty)
562: {
563:   spl_heap_object *intern = Z_SPLHEAP_P(getThis());
564:
565:   if (zend_parse_parameters_none() == FAILURE) {
566:     return;
567:   }
568:
569:   RETURN_BOOL(spl_ptr_heap_count(intern->heap) == 0);
570: }
571: /* }}} */
572:
573: /* {{{ proto bool SplHeap::insert(mixed value)
574:      Push $value on the heap */
575: SPL_METHOD(SplHeap, insert)
576: {
577:   zval *value;
578:   spl_heap_object *intern;
579:
580:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &value) == FAILURE) {
581:     return;
582:   }
583:
584:   intern = Z_SPLHEAP_P(getThis());
585:
586:   if (intern->heap->flags & SPL_HEAP_CORRUPTED) {
587:     zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
588:     return;
589:   }
590:
591:   Z_TRY_ADDREF_P(value);
592:   spl_ptr_heap_insert(intern->heap, value, getThis());
593:
594:   RETURN_TRUE;
595: }
596: /* }}} */
597:
598: /* {{{ proto mixed SplHeap::extract()
599:      extract the element out of the top of the heap */
600: SPL_METHOD(SplHeap, extract)
601: {
602:   spl_heap_object *intern;
603:
604:   if (zend_parse_parameters_none() == FAILURE) {
605:     return;
606:   }
607:
608:   intern = Z_SPLHEAP_P(getThis());
609:
610:   if (intern->heap->flags & SPL_HEAP_CORRUPTED) {
611:     zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
612:     return;
613:   }
614:
615:   spl_ptr_heap_delete_top(intern->heap, return_value, getThis());
616:
617:   if (Z_ISUNDEF_P(return_value)) {
618:     zend_throw_exception(spl_ce_RuntimeException, "Can't extract from an empty heap", 0);
619:     return;
620:   }
621: }
622: /* }}} */
623:
624: /* {{{ proto bool SplPriorityQueue::insert(mixed value, mixed priority)
625:      Push $value with the priority $priodiry on the priorityqueue */
626: SPL_METHOD(SplPriorityQueue, insert)
627: {
628:   zval *data, *priority, elem;
629:   spl_heap_object *intern;
630:
631:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &data, &priority) == FAILURE) {
632:     return;
633:   }
634:
635:   intern = Z_SPLHEAP_P(getThis());
636:
637:   if (intern->heap->flags & SPL_HEAP_CORRUPTED) {
638:     zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
639:     return;
640:   }
641:
642:   Z_TRY_ADDREF_P(data);
643:   Z_TRY_ADDREF_P(priority);
644:
645:   array_init(&elem);
646:   add_assoc_zval_ex(&elem, "data", sizeof("data") - 1, data);
647:   add_assoc_zval_ex(&elem, "priority", sizeof("priority") - 1, priority);
648:
649:   spl_ptr_heap_insert(intern->heap, &elem, getThis());
650:
651:   RETURN_TRUE;
652: }
653: /* }}} */
654:
655: /* {{{ proto mixed SplPriorityQueue::extract()
656:      extract the element out of the top of the priority queue */
657: SPL_METHOD(SplPriorityQueue, extract)
658: {
659:   zval value, *value_out;
660:   spl_heap_object *intern;
661:
662:   if (zend_parse_parameters_none() == FAILURE) {
663:     return;
664:   }
665:
666:   intern = Z_SPLHEAP_P(getThis());
667:
668:   if (intern->heap->flags & SPL_HEAP_CORRUPTED) {
669:     zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
670:     return;
671:   }
672:
673:   spl_ptr_heap_delete_top(intern->heap, &value, getThis());
674:
675:   if (Z_ISUNDEF(value)) {
676:     zend_throw_exception(spl_ce_RuntimeException, "Can't extract from an empty heap", 0);
677:     return;
678:   }
679:
680:   value_out = spl_pqueue_extract_helper(&value, intern->flags);
681:
682:   if (!value_out) {
683:     zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
684:     zval_ptr_dtor(&value);
685:     return;
686:   }
687:
688:   ZVAL_DEREF(value_out);
689:   ZVAL_COPY(return_value, value_out);
690:   zval_ptr_dtor(&value);
691: }
692: /* }}} */
693:
694: /* {{{ proto mixed SplPriorityQueue::top()
695:      Peek at the top element of the priority queue */
696: SPL_METHOD(SplPriorityQueue, top)
697: {
698:   zval *value, *value_out;
699:   spl_heap_object *intern;
700:
701:   if (zend_parse_parameters_none() == FAILURE) {
702:     return;
703:   }
704:
705:   intern = Z_SPLHEAP_P(getThis());
706:
707:   if (intern->heap->flags & SPL_HEAP_CORRUPTED) {
708:     zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
709:     return;
710:   }
711:
712:   value = spl_ptr_heap_top(intern->heap);
713:
714:   if (!value) {
715:     zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty heap", 0);
716:     return;
717:   }
718:
719:   value_out = spl_pqueue_extract_helper(value, intern->flags);
720:
721:   if (!value_out) {
722:     zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
723:     return;
724:   }
725:
726:   ZVAL_DEREF(value_out);
727:   ZVAL_COPY(return_value, value_out);
728: }
729: /* }}} */
730:
731:
732: /* {{{ proto int SplPriorityQueue::setExtractFlags(int flags)
733:  Set the flags of extraction*/
734: SPL_METHOD(SplPriorityQueue, setExtractFlags)
735: {
736:   zend_long value;
737:   spl_heap_object *intern;
738:
739:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &value) == FAILURE) {
740:     return;
741:   }
742:
743:   intern = Z_SPLHEAP_P(getThis());
744:
745:   intern->flags = value & SPL_PQUEUE_EXTR_MASK;
746:
747:   RETURN_LONG(intern->flags);
748: }
749: /* }}} */
750:
751: /* {{{ proto int SplPriorityQueue::getExtractFlags()
752:  Get the flags of extraction*/
```

```
753: SPL_METHOD(SplPriorityQueue, getExtractFlags)
754: {
755:   spl_heap_object *intern;
756:
757:   if (zend_parse_parameters_none() == FAILURE) {
758:     return;
759:   }
760:
761:   intern = Z_SPLHEAP_P(getThis());
762:
763:   RETURN_LONG(intern->flags);
764: }
765: /* }}} */
766:
767: /* {{{ proto int SplHeap::recoverFromCorruption()
768:   Recover from a corrupted state*/
769: SPL_METHOD(SplHeap, recoverFromCorruption)
770: {
771:   spl_heap_object *intern;
772:
773:   if (zend_parse_parameters_none() == FAILURE) {
774:     return;
775:   }
776:
777:   intern = Z_SPLHEAP_P(getThis());
778:
779:   intern->heap->flags = intern->heap->flags & ~SPL_HEAP_CORRUPTED;
780:
781:   RETURN_TRUE;
782: }
783: /* }}} */
784:
785: /* {{{ proto int SplHeap::isCorrupted()
786:   Tells if the heap is in a corrupted state*/
787: SPL_METHOD(SplHeap, isCorrupted)
788: {
789:   spl_heap_object *intern;
790:
791:   if (zend_parse_parameters_none() == FAILURE) {
792:     return;
793:   }
794:
795:   intern = Z_SPLHEAP_P(getThis());
796:
797:   RETURN_BOOL(intern->heap->flags & SPL_HEAP_CORRUPTED);
798: }
799: /* }}} */
800:
801: /* {{{ proto bool SplPriorityQueue::compare(mixed $a, mixed $b)
802:     compare the priorities */
803: SPL_METHOD(SplPriorityQueue, compare)
804: {
805:   zval *a, *b;
806:
807:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &a, &b) == FAILURE) {
808:     return;
809:   }
810:
811:   RETURN_LONG(spl_ptr_heap_zval_max_cmp(a, b, NULL));
812: }
813: /* }}} */
814:
815: /* {{{ proto mixed SplHeap::top()
816:     Peek at the top element of the heap */
817: SPL_METHOD(SplHeap, top)
818: {
819:   zval *value;
820:   spl_heap_object *intern;
821:
822:   if (zend_parse_parameters_none() == FAILURE) {
823:     return;
824:   }
825:
826:   intern = Z_SPLHEAP_P(getThis());
827:
828:   if (intern->heap->flags & SPL_HEAP_CORRUPTED) {
829:     zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
830:     return;
831:   }
832:
833:   value = spl_ptr_heap_top(intern->heap);
834:
835:   if (!value) {
836:     zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty heap", 0);
837:     return;
838:   }
839:
840:   ZVAL_DEREF(value);
841:   ZVAL_COPY(return_value, value);
842: }
843: /* }}} */
844:
845: /* {{{ proto bool SplMinHeap::compare(mixed $a, mixed $b)
846:     compare the values */
847: SPL_METHOD(SplMinHeap, compare)
848: {
849:   zval *a, *b;
850:
851:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &a, &b) == FAILURE) {
852:     return;
853:   }
854:
855:   RETURN_LONG(spl_ptr_heap_zval_min_cmp(a, b, NULL));
856: }
857: /* }}} */
858:
859: /* {{{ proto bool SplMaxHeap::compare(mixed $a, mixed $b)
860:     compare the values */
861: SPL_METHOD(SplMaxHeap, compare)
862: {
863:   zval *a, *b;
864:
865:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &a, &b) == FAILURE) {
866:     return;
867:   }
868:
869:   RETURN_LONG(spl_ptr_heap_zval_max_cmp(a, b, NULL));
870: }
871: /* }}} */
872:
873: static void spl_heap_it_dtor(zend_object_iterator *iter) /* {{{ */
874: {
875:   spl_heap_it *iterator = (spl_heap_it *)iter;
876:
877:   zend_user_it_invalidate_current(iter);
878:   zval_ptr_dtor(&iterator->intern.it.data);
879: }
880: /* }}} */
881:
882: static void spl_heap_it_rewind(zend_object_iterator *iter) /* {{{ */
883: {
884:   /* do nothing, the iterator always points to the top element */
885: }
886: /* }}} */
887:
888: static int spl_heap_it_valid(zend_object_iterator *iter) /* {{{ */
889: {
890:   return ((Z_SPLHEAP_P(&iter->data))->heap->count != 0 ? SUCCESS : FAILURE);
891: }
892: /* }}} */
893:
894: static zval *spl_heap_it_get_current_data(zend_object_iterator *iter) /* {{{ */
895: {
896:   spl_heap_object *object = Z_SPLHEAP_P(&iter->data);
897:   zval *element = &object->heap->elements[0];
898:
899:   if (object->heap->flags & SPL_HEAP_CORRUPTED) {
900:     zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
901:     return NULL;
902:   }
903:
904:   if (object->heap->count == 0 || Z_ISUNDEF_P(element)) {
905:     return NULL;
906:   } else {
907:     return element;
908:   }
909: }
910: /* }}} */
911:
912: static zval *spl_pqueue_it_get_current_data(zend_object_iterator *iter) /* {{{ */
913: {
914:   spl_heap_object *object = Z_SPLHEAP_P(&iter->data);
915:   zval *element = &object->heap->elements[0];
916:
917:   if (object->heap->flags & SPL_HEAP_CORRUPTED) {
918:     zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
919:     return NULL;
920:   }
921:
922:   if (object->heap->count == 0 || Z_ISUNDEF_P(element)) {
923:     return NULL;
924:   } else {
925:     zval *data = spl_pqueue_extract_helper(element, object->flags);
926:     if (!data) {
927:       zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
928:     }
929:     return data;
930:   }
931: }
932: /* }}} */
933:
934: static void spl_heap_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
935: {
936:   spl_heap_object *object = Z_SPLHEAP_P(&iter->data);
937:
938:   ZVAL_LONG(key, object->heap->count - 1);
939: }
940: /* }}} */
941:
942: static void spl_heap_it_move_forward(zend_object_iterator *iter) /* {{{ */
943: {
944:   spl_heap_object *object = Z_SPLHEAP_P(&iter->data);
945:   zval elem;
946:
947:   if (object->heap->flags & SPL_HEAP_CORRUPTED) {
948:     zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
949:     return;
950:   }
951:
952:   spl_ptr_heap_delete_top(object->heap, &elem, &iter->data);
953:
954:   zval_ptr_dtor(&elem);
955:
956:   zend_user_it_invalidate_current(iter);
957: }
958: /* }}} */
959:
960: /* {{{  proto int SplHeap::key()
961:   Return current array key */
962: SPL_METHOD(SplHeap, key)
963: {
964:   spl_heap_object *intern = Z_SPLHEAP_P(getThis());
965:
966:   if (zend_parse_parameters_none() == FAILURE) {
967:     return;
968:   }
969:
970:   RETURN_LONG(intern->heap->count - 1);
971: }
972: /* }}} */
973:
974: /* {{{ proto void SplHeap::next()
975:   Move to next entry */
976: SPL_METHOD(SplHeap, next)
977: {
978:   spl_heap_object *intern = Z_SPLHEAP_P(getThis());
979:   zval elem;
980:   spl_ptr_heap_delete_top(intern->heap, &elem, getThis());
981:
982:   if (zend_parse_parameters_none() == FAILURE) {
983:     return;
984:   }
985:
986:   zval_ptr_dtor(&elem);
987: }
988: /* }}} */
989:
990: /* {{{ proto bool SplHeap::valid()
991:   Check whether the datastructure contains more entries */
992: SPL_METHOD(SplHeap, valid)
993: {
994:   spl_heap_object *intern = Z_SPLHEAP_P(getThis());
995:
996:   if (zend_parse_parameters_none() == FAILURE) {
997:     return;
998:   }
999:
1000:   RETURN_BOOL(intern->heap->count != 0);
1001: }
1002: /* }}} */
1003:
1004: /* {{{ proto void SplHeap::rewind()
1005:   Rewind the datastructure back to the start */
1006: SPL_METHOD(SplHeap, rewind)
1007: {
1008:   if (zend_parse_parameters_none() == FAILURE) {
1009:     return;
1010:   }
1011:   /* do nothing, the iterator always points to the top element */
1012: }
1013: /* }}} */
1014:
1015: /* {{{ proto mixed|NULL SplHeap::current()
1016:   Return current datastructure entry */
1017: SPL_METHOD(SplHeap, current)
1018: {
1019:   spl_heap_object *intern  = Z_SPLHEAP_P(getThis());
1020:   zval *element = &intern->heap->elements[0];
1021:
1022:   if (zend_parse_parameters_none() == FAILURE) {
1023:     return;
1024:   }
1025:
1026:   if (!intern->heap->count || Z_ISUNDEF_P(element)) {
1027:     RETURN_NULL();
1028:   } else {
1029:     ZVAL_DEREF(element);
1030:     ZVAL_COPY(return_value, element);
1031:   }
1032: }
1033: /* }}} */
1034:
1035: /* {{{ proto mixed|NULL SplPriorityQueue::current()
1036:   Return current datastructure entry */
1037: SPL_METHOD(SplPriorityQueue, current)
1038: {
1039:   spl_heap_object  *intern  = Z_SPLHEAP_P(getThis());
1040:   zval *element = &intern->heap->elements[0];
1041:
1042:   if (zend_parse_parameters_none() == FAILURE) {
1043:     return;
1044:   }
1045:
1046:   if (!intern->heap->count || Z_ISUNDEF_P(element)) {
1047:     RETURN_NULL();
1048:   } else {
1049:     zval *data = spl_pqueue_extract_helper(element, intern->flags);
1050:
1051:     if (!data) {
1052:       zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
1053:       RETURN_NULL();
1054:     }
1055:
1056:     ZVAL_DEREF(data);
1057:     ZVAL_COPY(return_value, data);
1058:   }
1059: }
1060: /* }}} */
1061:
1062: /* iterator handler table */
1063: static const zend_object_iterator_funcs spl_heap_it_funcs = {
1064:   spl_heap_it_dtor,
1065:   spl_heap_it_valid,
1066:   spl_heap_it_get_current_data,
1067:   spl_heap_it_get_current_key,
1068:   spl_heap_it_move_forward,
1069:   spl_heap_it_rewind,
1070:   NULL
1071: };
1072:
1073: static const zend_object_iterator_funcs spl_pqueue_it_funcs = {
1074:   spl_heap_it_dtor,
1075:   spl_heap_it_valid,
1076:   spl_pqueue_it_get_current_data,
1077:   spl_heap_it_get_current_key,
1078:   spl_heap_it_move_forward,
1079:   spl_heap_it_rewind,
1080:   NULL
1081: };
1082:
1083: zend_object_iterator *spl_heap_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1084: {
1085:   spl_heap_it     *iterator;
1086:   spl_heap_object *heap_object = Z_SPLHEAP_P(object);
1087:
1088:   if (by_ref) {
1089:     zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1090:     return NULL;
1091:   }
1092:
1093:   iterator = emalloc(sizeof(spl_heap_it));
1094:
1095:   zend_iterator_init(&iterator->intern.it);
1096:
1097:   ZVAL_COPY(&iterator->intern.it.data, object);
1098:   iterator->intern.it.funcs = &spl_heap_it_funcs;
1099:   iterator->intern.ce       = ce;
1100:   iterator->flags           = heap_object->flags;
1101:   ZVAL_UNDEF(&iterator->intern.value);
1102:
1103:   return &iterator->intern.it;
1104: }
1105: /* }}} */
1106:
1107: zend_object_iterator *spl_pqueue_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1108: {
1109:   spl_heap_it     *iterator;
1110:   spl_heap_object *heap_object = Z_SPLHEAP_P(object);
1111:
1112:   if (by_ref) {
1113:     zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1114:     return NULL;
1115:   }
1116:
1117:   iterator = emalloc(sizeof(spl_heap_it));
1118:
1119:   zend_iterator_init((zend_object_iterator*)iterator);
1120:
1121:   ZVAL_COPY(&iterator->intern.it.data, object);
1122:   iterator->intern.it.funcs = &spl_pqueue_it_funcs;
1123:   iterator->intern.ce       = ce;
1124:   iterator->flags           = heap_object->flags;
1125:
1126:   ZVAL_UNDEF(&iterator->intern.value);
1127:
1128:   return &iterator->intern.it;
```

```
1129: }
1130: /* }}} */
1131:
1132: ZEND_BEGIN_ARG_INFO(arginfo_heap_insert, 0)
1133:   ZEND_ARG_INFO(0, value)
1134: ZEND_END_ARG_INFO()
1135:
1136: ZEND_BEGIN_ARG_INFO(arginfo_heap_compare, 0)
1137:   ZEND_ARG_INFO(0, a)
1138:   ZEND_ARG_INFO(0, b)
1139: ZEND_END_ARG_INFO()
1140:
1141: ZEND_BEGIN_ARG_INFO(arginfo_pqueue_insert, 0)
1142:   ZEND_ARG_INFO(0, value)
1143:   ZEND_ARG_INFO(0, priority)
1144: ZEND_END_ARG_INFO()
1145:
1146: ZEND_BEGIN_ARG_INFO(arginfo_pqueue_setflags, 0)
1147:   ZEND_ARG_INFO(0, flags)
1148: ZEND_END_ARG_INFO()
1149:
1150: ZEND_BEGIN_ARG_INFO(arginfo_splheap_void, 0)
1151: ZEND_END_ARG_INFO()
1152:
1153: static const zend_function_entry spl_funcs_SplMinHeap[] = {
1154:   SPL_ME(SplMinHeap, compare, arginfo_heap_compare, ZEND_ACC_PROTECTED)
1155:   PHP_FE_END
1156: };
1157: static const zend_function_entry spl_funcs_SplMaxHeap[] = {
1158:   SPL_ME(SplMaxHeap, compare, arginfo_heap_compare, ZEND_ACC_PROTECTED)
1159:   PHP_FE_END
1160: };
1161:
1162: static const zend_function_entry spl_funcs_SplPriorityQueue[] = {
1163:   SPL_ME(SplPriorityQueue, compare,           arginfo_heap_compare,    ZEND_ACC_PUBLIC)
1164:   SPL_ME(SplPriorityQueue, insert,            arginfo_pqueue_insert,   ZEND_ACC_PUBLIC)
1165:   SPL_ME(SplPriorityQueue, setExtractFlags,   arginfo_pqueue_setflags, ZEND_ACC_PUBLIC)
1166:   SPL_ME(SplPriorityQueue, getExtractFlags,   arginfo_splheap_void,    ZEND_ACC_PUBLIC)
1167:   SPL_ME(SplPriorityQueue, top,               arginfo_splheap_void,    ZEND_ACC_PUBLIC)
1168:   SPL_ME(SplPriorityQueue, extract,           arginfo_splheap_void,    ZEND_ACC_PUBLIC)
1169:   SPL_ME(SplHeap,          count,             arginfo_splheap_void,    ZEND_ACC_PUBLIC)
1170:   SPL_ME(SplHeap,          isEmpty,           arginfo_splheap_void,    ZEND_ACC_PUBLIC)
1171:   SPL_ME(SplHeap,          rewind,            arginfo_splheap_void,    ZEND_ACC_PUBLIC)
1172:   SPL_ME(SplPriorityQueue, current,           arginfo_splheap_void,    ZEND_ACC_PUBLIC)
1173:   SPL_ME(SplHeap,          key,               arginfo_splheap_void,    ZEND_ACC_PUBLIC)
1174:   SPL_ME(SplHeap,          next,              arginfo_splheap_void,    ZEND_ACC_PUBLIC)
1175:   SPL_ME(SplHeap,          valid,             arginfo_splheap_void,    ZEND_ACC_PUBLIC)
1176:   SPL_ME(SplHeap,          recoverFromCorruption, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1177:   SPL_ME(SplHeap,          isCorrupted,       arginfo_splheap_void,    ZEND_ACC_PUBLIC)
1178:   PHP_FE_END
1179: };
1180:
1181: static const zend_function_entry spl_funcs_SplHeap[] = {
1182:   SPL_ME(SplHeap, extract,               arginfo_splheap_void, ZEND_ACC_PUBLIC)
1183:   SPL_ME(SplHeap, insert,                arginfo_heap_insert, ZEND_ACC_PUBLIC)
1184:   SPL_ME(SplHeap, top,                   arginfo_splheap_void, ZEND_ACC_PUBLIC)
1185:   SPL_ME(SplHeap, count,                 arginfo_splheap_void, ZEND_ACC_PUBLIC)
1186:   SPL_ME(SplHeap, isEmpty,               arginfo_splheap_void, ZEND_ACC_PUBLIC)
1187:   SPL_ME(SplHeap, rewind,                arginfo_splheap_void, ZEND_ACC_PUBLIC)
1188:   SPL_ME(SplHeap, current,               arginfo_splheap_void, ZEND_ACC_PUBLIC)
1189:   SPL_ME(SplHeap, key,                   arginfo_splheap_void, ZEND_ACC_PUBLIC)
1190:   SPL_ME(SplHeap, next,                  arginfo_splheap_void, ZEND_ACC_PUBLIC)
1191:   SPL_ME(SplHeap, valid,                 arginfo_splheap_void, ZEND_ACC_PUBLIC)
1192:   SPL_ME(SplHeap, recoverFromCorruption, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1193:   SPL_ME(SplHeap, isCorrupted,           arginfo_splheap_void, ZEND_ACC_PUBLIC)
1194:   ZEND_FENTRY(compare, NULL, NULL, ZEND_ACC_PROTECTED|ZEND_ACC_ABSTRACT)
1195:   PHP_FE_END
1196: };
1197: /* }}} */
1198:
1199: PHP_MINIT_FUNCTION(spl_heap) /* {{{ */
1200: {
1201:   REGISTER_SPL_STD_CLASS_EX(SplHeap, spl_heap_object_new, spl_funcs_SplHeap);
1202:   memcpy(&spl_handler_SplHeap, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1203:
1204:   spl_handler_SplHeap.offset         = XtOffsetOf(spl_heap_object, std);
1205:   spl_handler_SplHeap.clone_obj      = spl_heap_object_clone;
1206:   spl_handler_SplHeap.count_elements = spl_heap_object_count_elements;
1207:   spl_handler_SplHeap.get_debug_info = spl_heap_object_get_debug_info;
1208:   spl_handler_SplHeap.get_gc         = spl_heap_object_get_gc;
1209:   spl_handler_SplHeap.dtor_obj = zend_objects_destroy_object;
1210:   spl_handler_SplHeap.free_obj = spl_heap_object_free_storage;
1211:
1212:   REGISTER_SPL_IMPLEMENTS(SplHeap, Iterator);
1213:   REGISTER_SPL_IMPLEMENTS(SplHeap, Countable);
1214:
1215:   spl_ce_SplHeap->get_iterator = spl_heap_get_iterator;
1216:
1217:   REGISTER_SPL_SUB_CLASS_EX(SplMinHeap,          SplHeap,        spl_heap_object_new, spl_funcs_SplMinHeap);
1218:   REGISTER_SPL_SUB_CLASS_EX(SplMaxHeap,          SplHeap,        spl_heap_object_new, spl_funcs_SplMaxHeap);
1219:
1220:   spl_ce_SplMaxHeap->get_iterator = spl_heap_get_iterator;
1221:   spl_ce_SplMinHeap->get_iterator = spl_heap_get_iterator;
1222:
1223:   REGISTER_SPL_STD_CLASS_EX(SplPriorityQueue, spl_heap_object_new, spl_funcs_SplPriorityQueue);
1224:   memcpy(&spl_handler_SplPriorityQueue, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1225:
1226:   spl_handler_SplPriorityQueue.offset         = XtOffsetOf(spl_heap_object, std);
1227:   spl_handler_SplPriorityQueue.clone_obj      = spl_heap_object_clone;
1228:   spl_handler_SplPriorityQueue.count_elements = spl_heap_object_count_elements;
1229:   spl_handler_SplPriorityQueue.get_debug_info = spl_pqueue_object_get_debug_info;
1230:   spl_handler_SplPriorityQueue.get_gc         = spl_heap_object_get_gc;
1231:   spl_handler_SplPriorityQueue.dtor_obj = zend_objects_destroy_object;
1232:   spl_handler_SplPriorityQueue.free_obj = spl_heap_object_free_storage;
1233:
1234:   REGISTER_SPL_IMPLEMENTS(SplPriorityQueue, Iterator);
1235:   REGISTER_SPL_IMPLEMENTS(SplPriorityQueue, Countable);
1236:
1237:   spl_ce_SplPriorityQueue->get_iterator = spl_pqueue_get_iterator;
1238:
1239:   REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_BOTH",     SPL_PQUEUE_EXTR_BOTH);
1240:   REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_PRIORITY", SPL_PQUEUE_EXTR_PRIORITY);
1241:   REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_DATA",     SPL_PQUEUE_EXTR_DATA);
1242:
1243:   return SUCCESS;
1244: }
1245: /* }}} */
1246:
1247: /*
1248:  * Local variables:
1249:  * tab-width: 4
1250:  * c-basic-offset: 4
1251:  * End:
1252:  * vim600: fdm=marker
1253:  * vim: noet sw=4 ts=4
1254:  */
1255:
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   |          Etienne Kneuss <colder@php.net>                             |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "ext/standard/php_array.h"
#include "ext/standard/php_var.h"
#include "zend_smart_str.h"
#include "zend_interfaces.h"
#include "zend_exceptions.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_observer.h"
#include "spl_iterators.h"
#include "spl_array.h"
#include "spl_exceptions.h"

SPL_METHOD(SplObserver, update);
SPL_METHOD(SplSubject, attach);
SPL_METHOD(SplSubject, detach);
SPL_METHOD(SplSubject, notify);

ZEND_BEGIN_ARG_INFO(arginfo_SplObserver_update, 0)
  ZEND_ARG_OBJ_INFO(0, SplSubject, SplSubject, 0)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_SplObserver[] = {
  SPL_ABSTRACT_ME(SplObserver, update,   arginfo_SplObserver_update)
  PHP_FE_END
};

ZEND_BEGIN_ARG_INFO(arginfo_SplSubject_attach, 0)
  ZEND_ARG_OBJ_INFO(0, SplObserver, SplObserver, 0)
ZEND_END_ARG_INFO();

ZEND_BEGIN_ARG_INFO(arginfo_SplSubject_void, 0)
ZEND_END_ARG_INFO();

/*ZEND_BEGIN_ARG_INFO_EX(arginfo_SplSubject_notify, 0, 0, 1)
  ZEND_ARG_OBJ_INFO(0, ignore, SplObserver, 1)
ZEND_END_ARG_INFO());*/

static const zend_function_entry spl_funcs_SplSubject[] = {
  SPL_ABSTRACT_ME(SplSubject,  attach,   arginfo_SplSubject_attach)
  SPL_ABSTRACT_ME(SplSubject,  detach,   arginfo_SplSubject_attach)
  SPL_ABSTRACT_ME(SplSubject,  notify,   arginfo_SplSubject_void)
  PHP_FE_END
};

PHPAPI zend_class_entry     *spl_ce_SplObserver;
PHPAPI zend_class_entry     *spl_ce_SplSubject;
PHPAPI zend_class_entry     *spl_ce_SplObjectStorage;
PHPAPI zend_class_entry     *spl_ce_MultipleIterator;

PHPAPI zend_object_handlers spl_handler_SplObjectStorage;

typedef struct _spl_SplObjectStorage { /* {{{ */
  HashTable         storage;
  zend_long         index;
  HashPosition      pos;
  zend_long         flags;
  zend_function    *fptr_get_hash;
  zval              *gcdata;
  size_t            gcdata_num;
  zend_object       std;
} spl_SplObjectStorage; /* }}} */

/* {{{ storage is an assoc array of [zend_object*]=>[zval *obj, zval *inf] */
typedef struct _spl_SplObjectStorageElement {
  zval obj;
  zval inf;
} spl_SplObjectStorageElement; /* }}} */

static inline spl_SplObjectStorage *spl_object_storage_from_obj(zend_object *obj) /* {{{ */ {
  return (spl_SplObjectStorage*)((char*)(obj) - XtOffsetOf(spl_SplObjectStorage, std));
}
/* }}} */

#define Z_SPLOBJSTORAGE_P(zv)  spl_object_storage_from_obj(Z_OBJ_P((zv)))

void spl_SplObjectStorage_free_storage(zend_object *object) /* {{{ */
{
  spl_SplObjectStorage *intern = spl_object_storage_from_obj(object);

  zend_object_std_dtor(&intern->std);

  zend_hash_destroy(&intern->storage);

  if (intern->gcdata != NULL) {
    efree(intern->gcdata);
  }

} /* }}} */

static int spl_object_storage_get_hash(zend_hash_key *key, spl_SplObjectStorage *intern, zval *this, zval *obj) {
  if (intern->fptr_get_hash) {
    zval rv;
    zend_call_method_with_1_params(this, intern->std.ce, &intern->fptr_get_hash, "getHash", &rv, obj);
    if (!Z_ISUNDEF(rv)) {
      if (Z_TYPE(rv) == IS_STRING) {
        key->key = Z_STR(rv);
        return SUCCESS;
      } else {
        zend_throw_exception(spl_ce_RuntimeException, "Hash needs to be a string", 0);

        zval_ptr_dtor(&rv);
        return FAILURE;
      }
    } else {
      return FAILURE;
    }
  } else {
    key->key = NULL;
    key->h = Z_OBJ_HANDLE_P(obj);
    return SUCCESS;
  }
}

static void spl_object_storage_free_hash(spl_SplObjectStorage *intern, zend_hash_key *key) {
  if (key->key) {
    zend_string_release(key->key);
  }
}

static void spl_object_storage_dtor(zval *element) /* {{{ */
{
  spl_SplObjectStorageElement *el = Z_PTR_P(element);
  zval_ptr_dtor(&el->obj);
  zval_ptr_dtor(&el->inf);
  efree(el);
} /* }}} */

static spl_SplObjectStorageElement* spl_object_storage_get(spl_SplObjectStorage *intern, zend_hash_key *key) /* {{{ */
{
  if (key->key) {
    return zend_hash_find_ptr(&intern->storage, key->key);
  } else {
    return zend_hash_index_find_ptr(&intern->storage, key->h);
  }
} /* }}} */

spl_SplObjectStorageElement *spl_object_storage_attach(spl_SplObjectStorage *intern, zval *this, zval *obj, zval *inf) /* {{{ */
{
  spl_SplObjectStorageElement *pelement, element;
  zend_hash_key key;
  if (spl_object_storage_get_hash(&key, intern, this, obj) == FAILURE) {
    return NULL;
  }

  pelement = spl_object_storage_get(intern, &key);

  if (pelement) {
    zval_ptr_dtor(&pelement->inf);
    if (inf) {
      ZVAL_COPY(&pelement->inf, inf);
    } else {
      ZVAL_NULL(&pelement->inf);
    }
    spl_object_storage_free_hash(intern, &key);
    return pelement;
  }

  ZVAL_COPY(&element.obj, obj);
  if (inf) {
    ZVAL_COPY(&element.inf, inf);
  } else {
    ZVAL_NULL(&element.inf);
  }
  if (key.key) {
    pelement = zend_hash_update_mem(&intern->storage, key.key, &element, sizeof(spl_SplObjectStorageElement));
  } else {
    pelement = zend_hash_index_update_mem(&intern->storage, key.h, &element, sizeof(spl_SplObjectStorageElement));
  }
  spl_object_storage_free_hash(intern, &key);
  return pelement;
} /* }}} */

static int spl_object_storage_detach(spl_SplObjectStorage *intern, zval *this, zval *obj) /* {{{ */
{
  int ret = FAILURE;
  zend_hash_key key;
  if (spl_object_storage_get_hash(&key, intern, this, obj) == FAILURE) {
    return ret;
  }
  if (key.key) {
    ret = zend_hash_del(&intern->storage, key.key);
  } else {
    ret = zend_hash_index_del(&intern->storage, key.h);
  }
  spl_object_storage_free_hash(intern, &key);

  return ret;
} /* }}} */

void spl_object_storage_addall(spl_SplObjectStorage *intern, zval *this, spl_SplObjectStorage *other) { /* {{{ */
  spl_SplObjectStorageElement *element;

  ZEND_HASH_FOREACH_PTR(&other->storage, element) {
    spl_object_storage_attach(intern, this, &element->obj, &element->inf);
  } ZEND_HASH_FOREACH_END();

  intern->index = 0;
} /* }}} */

static zend_object *spl_object_storage_new_ex(zend_class_entry *class_type, zval *orig) /* {{{ */
{
  spl_SplObjectStorage *intern;
  zend_class_entry *parent = class_type;

  intern = emalloc(sizeof(spl_SplObjectStorage) + zend_object_properties_size(parent));
  memset(intern, 0, sizeof(spl_SplObjectStorage) - sizeof(zval));
  intern->pos = HT_INVALID_IDX;

  zend_object_std_init(&intern->std, class_type);
  object_properties_init(&intern->std, class_type);

  zend_hash_init(&intern->storage, 0, NULL, spl_object_storage_dtor, 0);

  intern->std.handlers = &spl_handler_SplObjectStorage;

  while (parent) {
    if (parent == spl_ce_SplObjectStorage) {
      if (class_type != spl_ce_SplObjectStorage) {
        intern->fptr_get_hash = zend_hash_str_find_ptr(&class_type->function_table, "gethash", sizeof("gethash") - 1);
        if (intern->fptr_get_hash->common.scope == spl_ce_SplObjectStorage) {
          intern->fptr_get_hash = NULL;
        }
      }
      break;
    }

    parent = parent->parent;
  }

  if (orig) {
    spl_SplObjectStorage *other = Z_SPLOBJSTORAGE_P(orig);
    spl_object_storage_addall(intern, orig, other);
  }

  return &intern->std;
}
/* }}} */

/* {{{ spl_object_storage_clone */
static zend_object *spl_object_storage_clone(zval *zobject)
{
  zend_object *old_object;
  zend_object *new_object;

  old_object = Z_OBJ_P(zobject);
  new_object = spl_object_storage_new_ex(old_object->ce, zobject);

  zend_objects_clone_members(new_object, old_object);

  return new_object;
}
/* }}} */

static HashTable* spl_object_storage_debug_info(zval *obj, int *is_temp) /* {{{ */
{
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(obj);
  spl_SplObjectStorageElement *element;
  HashTable *props;
  zval tmp, storage;
  zend_string *md5str;
  zend_string *zname;
  HashTable *debug_info;

  *is_temp = 1;

  props = Z_OBJPROP_P(obj);

  debug_info = zend_new_array(zend_hash_num_elements(props) + 1);
  zend_hash_copy(debug_info, props, (copy_ctor_func_t)zval_add_ref);

  array_init(&storage);

  ZEND_HASH_FOREACH_PTR(&intern->storage, element) {
    md5str = php_spl_object_hash(&element->obj);
    array_init(&tmp);
    /* Incrementing the refcount of obj and inf would confuse the garbage collector.
     * Prefer to null the destructor */
    Z_ARRVAL_P(&tmp)->pDestructor = NULL;
    add_assoc_zval_ex(&tmp, "obj", sizeof("obj") - 1, &element->obj);
    add_assoc_zval_ex(&tmp, "inf", sizeof("inf") - 1, &element->inf);
    zend_hash_update(Z_ARRVAL(storage), md5str, &tmp);
    zend_string_release(md5str);
  } ZEND_HASH_FOREACH_END();

  zname = spl_gen_private_prop_name(spl_ce_SplObjectStorage, "storage", sizeof("storage")-1);
  zend_symtable_update(debug_info, zname, &storage);
  zend_string_release(zname);

  return debug_info;
}
/* }}} */

/* overriden for garbage collection */
static HashTable *spl_object_storage_get_gc(zval *obj, zval **table, int *n) /* {{{ */
{
  int i = 0;
  spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(obj);
  spl_SplObjectStorageElement *element;

  if (intern->storage.nNumOfElements * 2 > intern->gcdata_num) {
    intern->gcdata_num = intern->storage.nNumOfElements * 2;
    intern->gcdata = (zval*)erealloc(intern->gcdata, sizeof(zval) * intern->gcdata_num);
  }

  ZEND_HASH_FOREACH_PTR(&intern->storage, element) {
    ZVAL_COPY_VALUE(&intern->gcdata[i++], &element->obj);
    ZVAL_COPY_VALUE(&intern->gcdata[i++], &element->inf);
  } ZEND_HASH_FOREACH_END();

  *table = intern->gcdata;
  *n = i;

  return std_object_handlers.get_properties(obj);
}
/* }}} */

static int spl_object_storage_compare_info(zval *e1, zval *e2) /* {{{ */
{
  spl_SplObjectStorageElement *s1 = (spl_SplObjectStorageElement*)Z_PTR_P(e1);
  spl_SplObjectStorageElement *s2 = (spl_SplObjectStorageElement*)Z_PTR_P(e2);
  zval result;

  if (compare_function(&result, &s1->inf, &s2->inf) == FAILURE) {
    return 1;
  }

  return Z_LVAL(result) > 0 ? 1 : (Z_LVAL(result) < 0 ? -1 : 0);
}
/* }}} */

static int spl_object_storage_compare_objects(zval *o1, zval *o2) /* {{{ */
{
  zend_object *zo1 = (zend_object *)Z_OBJ_P(o1);
  zend_object *zo2 = (zend_object *)Z_OBJ_P(o2);

  if (zo1->ce != spl_ce_SplObjectStorage || zo2->ce != spl_ce_SplObjectStorage) {
    return 1;
  }

  return zend_hash_compare(&(Z_SPLOBJSTORAGE_P(o1))->storage, &(Z_SPLOBJSTORAGE_P(o2))->storage, (compare_func_t)spl_object_storage_compare_info, 0);
}
/* }}} */

/* {{{ spl_array_object_new */
static zend_object *spl_SplObjectStorage_new(zend_class_entry *class_type)
{
```

```
377:   return spl_object_storage_new_ex(class_type, NULL);
378: }
379: /* }}} */
380:
381: int spl_object_storage_contains(spl_SplObjectStorage *intern, zval *this, zval *obj) /* {{{ */
382: {
383:   int found;
384:   zend_hash_key key;
385:   if (spl_object_storage_get_hash(&key, intern, this, obj) == FAILURE) {
386:     return 0;
387:   }
388:
389:   if (key.key) {
390:     found = zend_hash_exists(&intern->storage, key.key);
391:   } else {
392:     found = zend_hash_index_exists(&intern->storage, key.h);
393:   }
394:   spl_object_storage_free_hash(intern, &key);
395:   return found;
396: } /* }}} */
397:
398: /* {{{ proto void SplObjectStorage::attach(object obj, mixed inf = NULL)
399:  Attaches an object to the storage if not yet contained */
400: SPL_METHOD(SplObjectStorage, attach)
401: {
402:   zval *obj, *inf = NULL;
403:
404:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
405:
406:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "o|z!", &obj, &inf) == FAILURE) {
407:     return;
408:   }
409:   spl_object_storage_attach(intern, getThis(), obj, inf);
410: } /* }}} */
411:
412: /* {{{ proto void SplObjectStorage::detach(object obj)
413:  Detaches an object from the storage */
414: SPL_METHOD(SplObjectStorage, detach)
415: {
416:   zval *obj;
417:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
418:
419:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
420:     return;
421:   }
422:   spl_object_storage_detach(intern, getThis(), obj);
423:
424:   zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
425:   intern->index = 0;
426: } /* }}} */
427:
428: /* {{{ proto string SplObjectStorage::getHash(object obj)
429:  Returns the hash of an object */
430: SPL_METHOD(SplObjectStorage, getHash)
431: {
432:   zval *obj;
433:
434:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
435:     return;
436:   }
437:
438:   RETURN_NEW_STR(php_spl_object_hash(obj));
439:
440: } /* }}} */
441:
442: /* {{{ proto mixed SplObjectStorage::offsetGet(object obj)
443:  Returns associated information for a stored object */
444: SPL_METHOD(SplObjectStorage, offsetGet)
445: {
446:   zval *obj;
447:   spl_SplObjectStorageElement *element;
448:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
449:   zend_hash_key key;
450:
451:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
452:     return;
453:   }
454:
455:   if (spl_object_storage_get_hash(&key, intern, getThis(), obj) == FAILURE) {
456:     return;
457:   }
458:
459:   element = spl_object_storage_get(intern, &key);
460:   spl_object_storage_free_hash(intern, &key);
461:
462:   if (!element) {
463:     zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Object not found");
464:   } else {
465:     zval *value = &element->inf;
466:
467:     ZVAL_DEREF(value);
468:     ZVAL_COPY(return_value, value);
469:   }
470: } /* }}} */
471:
472: /* {{{ proto bool SplObjectStorage::addAll(SplObjectStorage $os)
473:  Add all elements contained in $os */
474: SPL_METHOD(SplObjectStorage, addAll)
475: {
476:   zval *obj;
477:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
478:   spl_SplObjectStorage *other;
479:
480:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, spl_ce_SplObjectStorage) == FAILURE) {
481:     return;
482:   }
483:
484:   other = Z_SPLOBJSTORAGE_P(obj);
485:
486:   spl_object_storage_addall(intern, getThis(), other);
487:
488:   RETURN_LONG(zend_hash_num_elements(&intern->storage));
489: } /* }}} */
490:
491: /* {{{ proto bool SplObjectStorage::removeAll(SplObjectStorage $os)
492:  Remove all elements contained in $os */
493: SPL_METHOD(SplObjectStorage, removeAll)
494: {
495:   zval *obj;
496:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
497:   spl_SplObjectStorage *other;
498:   spl_SplObjectStorageElement *element;
499:
500:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, spl_ce_SplObjectStorage) == FAILURE) {
501:     return;
502:   }
503:
504:   other = Z_SPLOBJSTORAGE_P(obj);
505:
506:   zend_hash_internal_pointer_reset(&other->storage);
507:   while ((element = zend_hash_get_current_data_ptr(&other->storage)) != NULL) {
508:     if (spl_object_storage_detach(intern, getThis(), element->obj) == FAILURE) {
509:       zend_hash_move_forward(&other->storage);
510:     }
511:   }
512:
513:   zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
514:   intern->index = 0;
515:
516:   RETURN_LONG(zend_hash_num_elements(&intern->storage));
517: } /* }}} */
518:
519: /* {{{ proto bool SplObjectStorage::removeAllExcept(SplObjectStorage $os)
520:  Remove elements not common to both this SplObjectStorage instance and $os */
521: SPL_METHOD(SplObjectStorage, removeAllExcept)
522: {
523:   zval *obj;
524:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
525:   spl_SplObjectStorage *other;
526:   spl_SplObjectStorageElement *element;
527:
528:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, spl_ce_SplObjectStorage) == FAILURE) {
529:     return;
530:   }
531:
532:   other = Z_SPLOBJSTORAGE_P(obj);
533:
534:   ZEND_HASH_FOREACH_PTR(&intern->storage, element) {
535:     if (!spl_object_storage_contains(other, getThis(), &element->obj)) {
536:       spl_object_storage_detach(intern, getThis(), &element->obj);
537:     }
538:   } ZEND_HASH_FOREACH_END();
539:
540:   zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
541:   intern->index = 0;
542:
543:   RETURN_LONG(zend_hash_num_elements(&intern->storage));
544: }
545: /* }}} */
546:
547: /* {{{ proto bool SplObjectStorage::contains(object obj)
548:  Determine whethe an object is contained in the storage */
549: SPL_METHOD(SplObjectStorage, contains)
550: {
551:   zval *obj;
552:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
553:
554:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
555:     return;
556:   }
557:   RETURN_BOOL(spl_object_storage_contains(intern, getThis(), obj));
558: } /* }}} */
559:
560: /* {{{ proto int SplObjectStorage::count()
561:  Determine number of objects in storage */
562: SPL_METHOD(SplObjectStorage, count)
563: {
564:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
565:   zend_long mode = COUNT_NORMAL;
566:
567:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "|l", &mode) == FAILURE) {
568:     return;
569:   }
570:
571:   if (mode == COUNT_RECURSIVE) {
572:     zend_long ret;
573:
574:     if (mode != COUNT_RECURSIVE) {
575:       ret = zend_hash_num_elements(&intern->storage);
576:     } else {
577:       ret = php_count_recursive(&intern->storage);
578:     }
579:
580:     RETURN_LONG(ret);
581:     return;
582:   }
583:
584:   RETURN_LONG(zend_hash_num_elements(&intern->storage));
585: } /* }}} */
586:
587: /* {{{ proto void SplObjectStorage::rewind()
588:  Rewind to first position */
589: SPL_METHOD(SplObjectStorage, rewind)
590: {
591:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
592:
593:   if (zend_parse_parameters_none() == FAILURE) {
594:     return;
595:   }
596:
597:   zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
598:   intern->index = 0;
599: } /* }}} */
600:
601: /* {{{ proto bool SplObjectStorage::valid()
602:  Returns whether current position is valid */
603: SPL_METHOD(SplObjectStorage, valid)
604: {
605:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
606:
607:   if (zend_parse_parameters_none() == FAILURE) {
608:     return;
609:   }
610:
611:   RETURN_BOOL(zend_hash_has_more_elements_ex(&intern->storage, &intern->pos) == SUCCESS);
612: } /* }}} */
613:
614: /* {{{ proto mixed SplObjectStorage::key()
615:  Returns current key */
616: SPL_METHOD(SplObjectStorage, key)
617: {
618:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
619:
620:   if (zend_parse_parameters_none() == FAILURE) {
621:     return;
622:   }
623:
624:   RETURN_LONG(intern->index);
625: } /* }}} */
626:
627: /* {{{ proto mixed SplObjectStorage::current()
628:  Returns current element */
629: SPL_METHOD(SplObjectStorage, current)
630: {
631:   spl_SplObjectStorageElement *element;
632:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
633:
634:   if (zend_parse_parameters_none() == FAILURE) {
635:     return;
636:   }
637:
638:   if ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) == NULL) {
639:     return;
640:   }
641:   ZVAL_COPY(return_value, &element->obj);
642: } /* }}} */
643:
644: /* {{{ proto mixed SplObjectStorage::getInfo()
645:  Returns associated information to current element */
646: SPL_METHOD(SplObjectStorage, getInfo)
647: {
648:   spl_SplObjectStorageElement *element;
649:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
650:
651:   if (zend_parse_parameters_none() == FAILURE) {
652:     return;
653:   }
654:
655:   if ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) == NULL) {
656:     return;
657:   }
658:   ZVAL_COPY(return_value, &element->inf);
659: } /* }}} */
660:
661: /* {{{ proto mixed SplObjectStorage::setInfo(mixed $inf)
662:  Sets associated information of current element to $inf */
663: SPL_METHOD(SplObjectStorage, setInfo)
664: {
665:   spl_SplObjectStorageElement *element;
666:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
667:   zval *inf;
668:
669:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &inf) == FAILURE) {
670:     return;
671:   }
672:
673:   if ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) == NULL) {
674:     return;
675:   }
676:   zval_ptr_dtor(&element->inf);
677:   ZVAL_COPY(&element->inf, inf);
678: } /* }}} */
679:
680: /* {{{ proto void SplObjectStorage::next()
681:  Moves position forward */
682: SPL_METHOD(SplObjectStorage, next)
683: {
684:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
685:
686:   if (zend_parse_parameters_none() == FAILURE) {
687:     return;
688:   }
689:
690:   zend_hash_move_forward_ex(&intern->storage, &intern->pos);
691:   intern->index++;
692: } /* }}} */
693:
694: /* {{{ proto string SplObjectStorage::serialize()
695:  Serializes storage */
696: SPL_METHOD(SplObjectStorage, serialize)
697: {
698:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
699:
700:   spl_SplObjectStorageElement *element;
701:   zval members, flags;
702:   HashPosition       pos;
703:   php_serialize_data_t var_hash;
704:   smart_str buf = {0};
705:
706:   if (zend_parse_parameters_none() == FAILURE) {
707:     return;
708:   }
709:
710:   PHP_VAR_SERIALIZE_INIT(var_hash);
711:
712:   /* storage */
713:   smart_str_appendl(&buf, "x:", 2);
714:   ZVAL_LONG(&flags, zend_hash_num_elements(&intern->storage));
715:   php_var_serialize(&buf, &flags, &var_hash);
716:   zval_ptr_dtor(&flags);
717:
718:   zend_hash_internal_pointer_reset_ex(&intern->storage, &pos);
719:
720:   while (zend_hash_has_more_elements_ex(&intern->storage, &pos) == SUCCESS) {
721:     if ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &pos)) == NULL) {
722:       smart_str_free(&buf);
723:       PHP_VAR_SERIALIZE_DESTROY(var_hash);
724:       RETURN_NULL();
725:     }
726:     php_var_serialize(&buf, &element->obj, &var_hash);
727:     smart_str_appendc(&buf, ',');
728:     php_var_serialize(&buf, &element->inf, &var_hash);
729:     smart_str_appendc(&buf, ';');
730:     zend_hash_move_forward_ex(&intern->storage, &pos);
731:   }
732:
733:   /* members */
734:   smart_str_appendl(&buf, "m:", 2);
735:
736:   ZVAL_ARR(&members, zend_array_dup(zend_std_get_properties(getThis())));
737:   php_var_serialize(&buf, &members, &var_hash); /* finishes the string */
738:   zval_ptr_dtor(&members);
739:
740:   /* done */
741:   PHP_VAR_SERIALIZE_DESTROY(var_hash);
742:
743:   if (buf.s) {
744:     RETURN_NEW_STR(buf.s);
745:   } else {
746:     RETURN_NULL();
747:   }
748:
749: } /* }}} */
750:
751: /* {{{ proto void SplObjectStorage::unserialize(string serialized)
752:  Unserializes storage */
```

```
753: SPL_METHOD(SplObjectStorage, unserialize)
754: {
755:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
756:
757:   char *buf;
758:   size_t buf_len;
759:   const unsigned char *p, *s;
760:   php_unserialize_data_t var_hash;
761:   zval entry, inf;
762:   zval *pcount, *pmembers;
763:   spl_SplObjectStorageElement *element;
764:   zend_long count;
765:
766:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &buf, &buf_len) == FAILURE) {
767:     return;
768:   }
769:
770:   if (buf_len == 0) {
771:     return;
772:   }
773:
774:   /* storage */
775:   s = p = (const unsigned char*)buf;
776:   PHP_VAR_UNSERIALIZE_INIT(var_hash);
777:
778:   if (*p!= 'x' || *++p != ':') {
779:     goto outexcept;
780:   }
781:   ++p;
782:
783:   pcount = var_tmp_var(&var_hash);
784:   if (!php_var_unserialize(pcount, &p, s + buf_len, &var_hash) || Z_TYPE_P(pcount) != IS_LONG) {
785:     goto outexcept;
786:   }
787:
788:   --p; /* for ';' */
789:   count = Z_LVAL_P(pcount);
790:
791:   ZVAL_UNDEF(&entry);
792:   ZVAL_UNDEF(&inf);
793:
794:   while (count-- > 0) {
795:     spl_SplObjectStorageElement *pelement;
796:     zend_hash_key key;
797:
798:     if (*p != ';') {
799:       goto outexcept;
800:     }
801:     ++p;
802:     if(*p != 'O' && *p != 'C' && *p != 'r') {
803:       goto outexcept;
804:     }
805:     /* store reference to allow cross-references between different elements */
806:     if (!php_var_unserialize(&entry, &p, s + buf_len, &var_hash)) {
807:       goto outexcept;
808:     }
809:     if (*p == ',') { /* new version has inf */
810:       ++p;
811:       if (!php_var_unserialize(&inf, &p, s + buf_len, &var_hash)) {
812:         zval_ptr_dtor(&entry);
813:         goto outexcept;
814:       }
815:     }
816:     if (Z_TYPE(entry) != IS_OBJECT) {
817:       zval_ptr_dtor(&entry);
818:       zval_ptr_dtor(&inf);
819:       goto outexcept;
820:     }
821:
822:     if (spl_object_storage_get_hash(&key, intern, getThis(), &entry) == FAILURE) {
823:       zval_ptr_dtor(&entry);
824:       zval_ptr_dtor(&inf);
825:       goto outexcept;
826:     }
827:     pelement = spl_object_storage_get(intern, &key);
828:     spl_object_storage_free_hash(intern, &key);
829:     if (pelement) {
830:       if (!Z_ISUNDEF(pelement->inf)) {
831:         var_push_dtor(&var_hash, &pelement->inf);
832:       }
833:       if (!Z_ISUNDEF(pelement->obj)) {
834:         var_push_dtor(&var_hash, &pelement->obj);
835:       }
836:     }
837:     element = spl_object_storage_attach(intern, getThis(), &entry, Z_ISUNDEF(inf)?NULL:&inf);
838:     var_replace(&var_hash, &entry, &element->obj);
839:     var_replace(&var_hash, &inf, &element->inf);
840:     zval_ptr_dtor(&entry);
841:     ZVAL_UNDEF(&entry);
842:     zval_ptr_dtor(&inf);
843:     ZVAL_UNDEF(&inf);
844:   }
845:
846:   if (*p != ';') {
847:     goto outexcept;
848:   }
849:   ++p;
850:
851:   /* members */
852:   if (*p!= 'm' || *++p != ':') {
853:     goto outexcept;
854:   }
855:   ++p;
856:
857:   pmembers = var_tmp_var(&var_hash);
858:   if (!php_var_unserialize(pmembers, &p, s + buf_len, &var_hash) || Z_TYPE_P(pmembers) != IS_ARRAY) {
859:     goto outexcept;
860:   }
861:
862:   /* copy members */
863:   object_properties_load(&intern->std, Z_ARRVAL_P(pmembers));
864:
865:   PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
866:   return;
867:
868: outexcept:
869:   PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
870:   zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Error at offset %zd of %zd bytes", ((char*)p - buf), buf_len);
871:   return;
872:
873: } /* }}} */
874:
875: ZEND_BEGIN_ARG_INFO(arginfo_Object, 0)
876:   ZEND_ARG_INFO(0, object)
877: ZEND_END_ARG_INFO();
878:
879: ZEND_BEGIN_ARG_INFO_EX(arginfo_attach, 0, 0, 1)
880:   ZEND_ARG_INFO(0, object)
881:   ZEND_ARG_INFO(0, inf)
882: ZEND_END_ARG_INFO();
883:
884: ZEND_BEGIN_ARG_INFO(arginfo_Serialized, 0)
885:   ZEND_ARG_INFO(0, serialized)
886: ZEND_END_ARG_INFO();
887:
888: ZEND_BEGIN_ARG_INFO(arginfo_setInfo, 0)
889:   ZEND_ARG_INFO(0, info)
890: ZEND_END_ARG_INFO();
891:
892: ZEND_BEGIN_ARG_INFO(arginfo_getHash, 0)
893:   ZEND_ARG_INFO(0, object)
894: ZEND_END_ARG_INFO();
895:
896: ZEND_BEGIN_ARG_INFO_EX(arginfo_offsetGet, 0, 0, 1)
897:   ZEND_ARG_INFO(0, object)
898: ZEND_END_ARG_INFO()
899:
900: ZEND_BEGIN_ARG_INFO(arginfo_splobject_void, 0)
901: ZEND_END_ARG_INFO()
902:
903: static const zend_function_entry spl_funcs_SplObjectStorage[] = {
904:   SPL_ME(SplObjectStorage,  attach,     arginfo_attach,     0)
905:   SPL_ME(SplObjectStorage,  detach,     arginfo_Object,     0)
906:   SPL_ME(SplObjectStorage,  contains,   arginfo_Object,     0)
907:   SPL_ME(SplObjectStorage,  addAll,     arginfo_Object,     0)
908:   SPL_ME(SplObjectStorage,  removeAll,  arginfo_Object,     0)
909:   SPL_ME(SplObjectStorage,  removeAllExcept,  arginfo_Object,  0)
910:   SPL_ME(SplObjectStorage,  getInfo,    arginfo_splobject_void,0)
911:   SPL_ME(SplObjectStorage,  setInfo,    arginfo_setInfo,    0)
912:   SPL_ME(SplObjectStorage,  getHash,    arginfo_getHash,    0)
913:   /* Countable */
914:   SPL_ME(SplObjectStorage,  count,      arginfo_splobject_void,0)
915:   /* Iterator */
916:   SPL_ME(SplObjectStorage,  rewind,     arginfo_splobject_void,0)
917:   SPL_ME(SplObjectStorage,  valid,      arginfo_splobject_void,0)
918:   SPL_ME(SplObjectStorage,  key,        arginfo_splobject_void,0)
919:   SPL_ME(SplObjectStorage,  current,    arginfo_splobject_void,0)
920:   SPL_ME(SplObjectStorage,  next,       arginfo_splobject_void,0)
921:   /* Serializable */
922:   SPL_ME(SplObjectStorage,  unserialize, arginfo_Serialized,    0)
923:   SPL_ME(SplObjectStorage,  serialize,  arginfo_splobject_void,0)
924:   /* ArrayAccess */
925:   SPL_MA(SplObjectStorage, offsetExists, SplObjectStorage, contains, arginfo_offsetGet, 0)
926:   SPL_MA(SplObjectStorage, offsetSet,    SplObjectStorage, attach,   arginfo_attach, 0)
927:   SPL_MA(SplObjectStorage, offsetUnset,  SplObjectStorage, detach,   arginfo_offsetGet, 0)
928:   SPL_ME(SplObjectStorage, offsetGet,    arginfo_offsetGet,     0)
929:   PHP_FE_END
930: };
931:
932: typedef enum {
933:   MIT_NEED_ANY    = 0,
934:   MIT_NEED_ALL    = 1,
935:   MIT_KEYS_NUMERIC = 0,
936:   MIT_KEYS_ASSOC  = 2
937: } MultipleIteratorFlags;
938:
939: #define SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT   1
940: #define SPL_MULTIPLE_ITERATOR_GET_ALL_KEY       2
941:
942: /* {{{ proto void MultipleIterator::__construct([int flags = MIT_NEED_ALL|MIT_KEYS_NUMERIC])
943:    Iterator that iterates over several iterators one after the other */
944: SPL_METHOD(MultipleIterator, __construct)
945: {
946:   spl_SplObjectStorage   *intern;
947:   zend_long               flags = MIT_NEED_ALL|MIT_KEYS_NUMERIC;
948:
949:   if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "|l", &flags) == FAILURE) {
950:     return;
951:   }
952:
953:   intern = Z_SPLOBJSTORAGE_P(getThis());
954:   intern->flags = flags;
955: }
956: /* }}} */
957:
958: /* {{{ proto int MultipleIterator::getFlags()
959:    Return current flags */
960: SPL_METHOD(MultipleIterator, getFlags)
961: {
962:   spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
963:
964:   if (zend_parse_parameters_none() == FAILURE) {
965:     return;
966:   }
967:   RETURN_LONG(intern->flags);
968: }
969: /* }}} */
970:
971: /* {{{ proto int MultipleIterator::setFlags(int flags)
972:    Set flags */
973: SPL_METHOD(MultipleIterator, setFlags)
974: {
975:   spl_SplObjectStorage *intern;
976:   intern = Z_SPLOBJSTORAGE_P(getThis());
977:
978:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &intern->flags) == FAILURE) {
979:     return;
980:   }
981: }
982: /* }}} */
983:
984: /* {{{ proto void attachIterator(Iterator iterator[, mixed info]) throws InvalidArgumentException
985:    Attach a new iterator */
986: SPL_METHOD(MultipleIterator, attachIterator)
987: {
988:   spl_SplObjectStorage        *intern;
989:   zval                        *iterator = NULL, *info = NULL;
990:
991:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "O|z!", &iterator, zend_ce_iterator, &info) == FAILURE) {
992:     return;
993:   }
994:
995:   intern = Z_SPLOBJSTORAGE_P(getThis());
996:
997:   if (info != NULL) {
998:     spl_SplObjectStorageElement *element;
999:
1000:    if (Z_TYPE_P(info) != IS_LONG && Z_TYPE_P(info) != IS_STRING) {
1001:      zend_throw_exception(spl_ce_InvalidArgumentException, "Info must be NULL, integer or string", 0);
1002:      return;
1003:    }
1004:
1005:    zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1006:    while ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) != NULL) {
1007:      if (fast_is_identical_function(info, element->inf)) {
1008:        zend_throw_exception(spl_ce_InvalidArgumentException, "Key duplication error", 0);
1009:        return;
1010:      }
1011:      zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1012:    }
1013:  }
1014:
1015:  spl_object_storage_attach(intern, getThis(), iterator, info);
1016: }
1017: /* }}} */
1018:
1019: /* {{{ proto void MultipleIterator::rewind()
1020:    Rewind all attached iterator instances */
1021: SPL_METHOD(MultipleIterator, rewind)
1022: {
1023:   spl_SplObjectStorage        *intern;
1024:   spl_SplObjectStorageElement *element;
1025:   zval                        *it;
1026:
1027:   intern = Z_SPLOBJSTORAGE_P(getThis());
1028:
1029:   if (zend_parse_parameters_none() == FAILURE) {
1030:     return;
1031:   }
1032:
1033:   zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1034:   while ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) != NULL && !EG(exception)) {
1035:     it = &element->obj;
1036:     zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.zf_rewind, "rewind", NULL);
1037:     zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1038:   }
1039: }
1040: /* }}} */
1041:
1042: /* {{{ proto void MultipleIterator::next()
1043:    Move all attached iterator instances forward */
1044: SPL_METHOD(MultipleIterator, next)
1045: {
1046:   spl_SplObjectStorage        *intern;
1047:   spl_SplObjectStorageElement *element;
1048:   zval                        *it;
1049:
1050:   intern = Z_SPLOBJSTORAGE_P(getThis());
1051:
1052:   if (zend_parse_parameters_none() == FAILURE) {
1053:     return;
1054:   }
1055:
1056:   zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1057:   while ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) != NULL && !EG(exception)) {
1058:     it = &element->obj;
1059:     zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.zf_next, "next", NULL);
1060:     zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1061:   }
1062: }
1063: /* }}} */
1064:
1065: /* {{{ proto bool MultipleIterator::valid()
1066:    Return whether all or one sub iterator is valid depending on flags */
1067: SPL_METHOD(MultipleIterator, valid)
1068: {
1069:   spl_SplObjectStorage        *intern;
1070:   spl_SplObjectStorageElement *element;
1071:   zval                        *it, retval;
1072:   zend_long                    expect, valid;
1073:
1074:   intern = Z_SPLOBJSTORAGE_P(getThis());
1075:
1076:   if (zend_parse_parameters_none() == FAILURE) {
1077:     return;
1078:   }
1079:
1080:   if (!zend_hash_num_elements(&intern->storage)) {
1081:     RETURN_FALSE;
1082:   }
1083:
1084:   expect = (intern->flags & MIT_NEED_ALL) ? 1 : 0;
1085:
1086:   zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1087:   while ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) != NULL && !EG(exception)) {
1088:     it = &element->obj;
1089:     zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.zf_valid, "valid", &retval);
1090:
1091:     if (!Z_ISUNDEF(retval)) {
1092:       valid = (Z_TYPE(retval) == IS_TRUE);
1093:       zval_ptr_dtor(&retval);
1094:     } else {
1095:       valid = 0;
1096:     }
1097:
1098:     if (expect != valid) {
1099:       RETURN_BOOL(!expect);
1100:     }
1101:
1102:     zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1103:   }
1104:
1105:   RETURN_BOOL(expect);
1106: }
1107: /* }}} */
1108:
1109: static void spl_multiple_iterator_get_all(spl_SplObjectStorage *intern, int get_type, zval *return_value) /* {{{ */
1110: {
1111:   spl_SplObjectStorageElement *element;
1112:   zval                        *it, retval;
1113:   int                          valid = 1, num_elements;
1114:
1115:   num_elements = zend_hash_num_elements(&intern->storage);
1116:   if (num_elements < 1) {
1117:     RETURN_FALSE;
1118:   }
1119:
1120:   array_init_size(return_value, num_elements);
1121:
1122:   zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1123:   while ((element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos)) != NULL && !EG(exception)) {
1124:     it = &element->obj;
1125:     zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.zf_valid, "valid", &retval);
1126:
1127:     if (!Z_ISUNDEF(retval)) {
1128:       valid = Z_TYPE(retval) == IS_TRUE;
```

```
1129:       zval_ptr_dtor(&retval);
1130:     } else {
1131:       valid = 0;
1132:     }
1133:
1134:     if (valid) {
1135:       if (SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT == get_type) {
1136:         zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.zf_current, "current", &retval);
1137:       } else {
1138:         zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs.zf_key, "key", &retval);
1139:       }
1140:       if (Z_ISUNDEF(retval)) {
1141:         zend_throw_exception(spl_ce_RuntimeException, "Failed to call sub iterator method", 0);
1142:         return;
1143:       }
1144:     } else if (intern->flags & MIT_NEED_ALL) {
1145:       if (SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT == get_type) {
1146:         zend_throw_exception(spl_ce_RuntimeException, "Called current() with non valid sub iterator", 0);
1147:       } else {
1148:         zend_throw_exception(spl_ce_RuntimeException, "Called key() with non valid sub iterator", 0);
1149:       }
1150:       return;
1151:     } else {
1152:       ZVAL_NULL(&retval);
1153:     }
1154:
1155:     if (intern->flags & MIT_KEYS_ASSOC) {
1156:       switch (Z_TYPE(element->inf)) {
1157:         case IS_LONG:
1158:           add_index_zval(return_value, Z_LVAL(element->inf), &retval);
1159:           break;
1160:         case IS_STRING:
1161:           zend_symtable_update(Z_ARRVAL_P(return_value), Z_STR(element->inf), &retval);
1162:           break;
1163:         default:
1164:           zval_ptr_dtor(&retval);
1165:           zend_throw_exception(spl_ce_InvalidArgumentException, "Sub-Iterator is associated with NULL", 0);
1166:           return;
1167:       }
1168:     } else {
1169:       add_next_index_zval(return_value, &retval);
1170:     }
1171:
1172:     zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1173:   }
1174: }
1175: /* }}} */
1176:
1177: /* {{{ proto array current() throws RuntimeException throws InvalidArgumentException
1178:    Return an array of all registered Iterator instances current() result */
1179: SPL_METHOD(MultipleIterator, current)
1180: {
1181:   spl_SplObjectStorage       *intern;
1182:   intern = Z_SPLOBJSTORAGE_P(getThis());
1183:
1184:   if (zend_parse_parameters_none() == FAILURE) {
1185:     return;
1186:   }
1187:
1188:   spl_multiple_iterator_get_all(intern, SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT, return_value);
1189: }
1190: /* }}} */
1191:
1192: /* {{{ proto array MultipleIterator::key()
1193:    Return an array of all registered Iterator instances key() result */
1194: SPL_METHOD(MultipleIterator, key)
1195: {
1196:   spl_SplObjectStorage *intern;
1197:   intern = Z_SPLOBJSTORAGE_P(getThis());
1198:
1199:   if (zend_parse_parameters_none() == FAILURE) {
1200:     return;
1201:   }
1202:
1203:   spl_multiple_iterator_get_all(intern, SPL_MULTIPLE_ITERATOR_GET_ALL_KEY, return_value);
1204: }
1205: /* }}} */
1206:
1207: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_attachIterator, 0, 0, 1)
1208:   ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
1209:   ZEND_ARG_INFO(0, infos)
1210: ZEND_END_ARG_INFO();
1211:
1212: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_detachIterator, 0, 0, 1)
1213:   ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
1214: ZEND_END_ARG_INFO();
1215:
1216: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_containsIterator, 0, 0, 1)
1217:   ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
1218: ZEND_END_ARG_INFO();
1219:
1220: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_setflags, 0, 0, 1)
1221:   ZEND_ARG_INFO(0, flags)
1222: ZEND_END_ARG_INFO();
1223:
1224: static const zend_function_entry spl_funcs_MultipleIterator[] = {
1225:   SPL_ME(MultipleIterator,  __construct,          arginfo_MultipleIterator_setflags,          0)
1226:   SPL_ME(MultipleIterator,  getFlags,             arginfo_splobject_void,                     0)
1227:   SPL_ME(MultipleIterator,  setFlags,             arginfo_MultipleIterator_setflags,          0)
1228:   SPL_ME(MultipleIterator,  attachIterator,       arginfo_MultipleIterator_attachIterator,    0)
1229:   SPL_MA(MultipleIterator,  detachIterator,       SplObjectStorage, detach,   arginfo_MultipleIterator_detachIterator,   0)
1230:   SPL_MA(MultipleIterator,  containsIterator,     SplObjectStorage, contains, arginfo_MultipleIterator_containsIterator, 0)
1231:   SPL_MA(MultipleIterator,  countIterators,       SplObjectStorage, count,    arginfo_splobject_void,                    0)
1232:   /* Iterator */
1233:   SPL_ME(MultipleIterator,  rewind,               arginfo_splobject_void,                     0)
1234:   SPL_ME(MultipleIterator,  valid,                arginfo_splobject_void,                     0)
1235:   SPL_ME(MultipleIterator,  key,                  arginfo_splobject_void,                     0)
1236:   SPL_ME(MultipleIterator,  current,              arginfo_splobject_void,                     0)
1237:   SPL_ME(MultipleIterator,  next,                 arginfo_splobject_void,                     0)
1238:   PHP_FE_END
1239: };
1240:
1241: /* {{{ PHP_MINIT_FUNCTION(spl_observer) */
1242: PHP_MINIT_FUNCTION(spl_observer)
1243: {
1244:   REGISTER_SPL_INTERFACE(SplObserver);
1245:   REGISTER_SPL_INTERFACE(SplSubject);
1246:
1247:   REGISTER_SPL_STD_CLASS_EX(SplObjectStorage, spl_SplObjectStorage_new, spl_funcs_SplObjectStorage);
1248:   memcpy(&spl_handler_SplObjectStorage, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1249:
1250:   spl_handler_SplObjectStorage.offset          = XtOffsetOf(spl_SplObjectStorage, std);
1251:   spl_handler_SplObjectStorage.get_debug_info  = spl_object_storage_debug_info;
1252:   spl_handler_SplObjectStorage.compare_objects = spl_object_storage_compare_objects;
1253:   spl_handler_SplObjectStorage.clone_obj       = spl_object_storage_clone;
1254:   spl_handler_SplObjectStorage.get_gc          = spl_object_storage_get_gc;
1255:   spl_handler_SplObjectStorage.dtor_obj        = zend_objects_destroy_object;
1256:   spl_handler_SplObjectStorage.free_obj        = spl_SplObjectStorage_free_storage;
1257:
1258:   REGISTER_SPL_IMPLEMENTS(SplObjectStorage, Countable);
1259:   REGISTER_SPL_IMPLEMENTS(SplObjectStorage, Iterator);
1260:   REGISTER_SPL_IMPLEMENTS(SplObjectStorage, Serializable);
1261:   REGISTER_SPL_IMPLEMENTS(SplObjectStorage, ArrayAccess);
1262:
1263:   REGISTER_SPL_STD_CLASS_EX(MultipleIterator, spl_SplObjectStorage_new, spl_funcs_MultipleIterator);
1264:   REGISTER_SPL_ITERATOR(MultipleIterator);
1265:
1266:   REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_NEED_ANY",     MIT_NEED_ANY);
1267:   REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_NEED_ALL",     MIT_NEED_ALL);
1268:   REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_KEYS_NUMERIC", MIT_KEYS_NUMERIC);
1269:   REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_KEYS_ASSOC",   MIT_KEYS_ASSOC);
1270:
1271:   return SUCCESS;
1272: }
1273: /* }}} */
1274:
1275: /*
1276:  * Local variables:
1277:  * tab-width: 4
1278:  * c-basic-offset: 4
1279:  * End:
1280:  * vim600: fdm=marker
1281:  * vim: noet sw=4 ts=4
1282:  */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Etienne Kneuss <colder@php.net>                             |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "zend_exceptions.h"
#include "zend_hash.h"

#include "php_spl.h"
#include "ext/standard/info.h"
#include "ext/standard/php_var.h"
#include "zend_smart_str.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_iterators.h"
#include "spl_dllist.h"
#include "spl_exceptions.h"

zend_object_handlers spl_handler_SplDoublyLinkedList;
PHPAPI zend_class_entry  *spl_ce_SplDoublyLinkedList;
PHPAPI zend_class_entry  *spl_ce_SplQueue;
PHPAPI zend_class_entry  *spl_ce_SplStack;

#define SPL_LLIST_DELREF(elem) if(!--(elem)->rc) { \
	efree(elem); \
}

#define SPL_LLIST_CHECK_DELREF(elem) if((elem) && !--(elem)->rc) { \
	efree(elem); \
}

#define SPL_LLIST_ADDREF(elem) (elem)->rc++
#define SPL_LLIST_CHECK_ADDREF(elem) if(elem) (elem)->rc++

#define SPL_DLLIST_IT_DELETE 0x00000001 /* Delete flag makes the iterator delete the current element on next */
#define SPL_DLLIST_IT_LIFO   0x00000002 /* LIFO flag makes the iterator traverse the structure as a LastInFirstOut */
#define SPL_DLLIST_IT_MASK   0x00000003 /* Mask to isolate flags related to iterators */
#define SPL_DLLIST_IT_FIX    0x00000004 /* Backward/Forward bit is fixed */

#ifdef accept
#undef accept
#endif

typedef struct _spl_ptr_llist_element {
	struct _spl_ptr_llist_element *prev;
	struct _spl_ptr_llist_element *next;
	int                            rc;
	zval                           data;
} spl_ptr_llist_element;

typedef void (*spl_ptr_llist_dtor_func)(spl_ptr_llist_element *);
typedef void (*spl_ptr_llist_ctor_func)(spl_ptr_llist_element *);

typedef struct _spl_ptr_llist {
	spl_ptr_llist_element   *head;
	spl_ptr_llist_element   *tail;
	spl_ptr_llist_dtor_func  dtor;
	spl_ptr_llist_ctor_func  ctor;
	int count;
} spl_ptr_llist;

typedef struct _spl_dllist_object spl_dllist_object;
typedef struct _spl_dllist_it spl_dllist_it;

struct _spl_dllist_object {
	spl_ptr_llist        *llist;
	int                   traverse_position;
	spl_ptr_llist_element *traverse_pointer;
	int                   flags;
	zend_function         *fptr_offset_get;
	zend_function         *fptr_offset_set;
	zend_function         *fptr_offset_has;
	zend_function         *fptr_offset_del;
	zend_function         *fptr_count;
	zend_class_entry      *ce_get_iterator;
	zval                  *gc_data;
	int                   gc_data_count;
	zend_object           std;
};

/* define an overloaded iterator structure */
struct _spl_dllist_it {
	zend_user_iterator     intern;
	spl_ptr_llist_element *traverse_pointer;
	int                    traverse_position;
	int                    flags;
};

static inline spl_dllist_object *spl_dllist_from_obj(zend_object *obj) /* {{{ */ {
	return (spl_dllist_object*)((char*)(obj) - XtOffsetOf(spl_dllist_object, std));
}
/* }}} */

#define Z_SPLDLLIST_P(zv)  spl_dllist_from_obj(Z_OBJ_P((zv)))

/* {{{  spl_ptr_llist */
static void spl_ptr_llist_zval_dtor(spl_ptr_llist_element *elem) { /* {{{ */
	if (!Z_ISUNDEF(elem->data)) {
		zval_ptr_dtor(&elem->data);
		ZVAL_UNDEF(&elem->data);
	}
}
/* }}} */

static void spl_ptr_llist_zval_ctor(spl_ptr_llist_element *elem) { /* {{{ */
	if (Z_REFCOUNTED(elem->data)) {
		Z_ADDREF(elem->data);
	}
}
/* }}} */

static spl_ptr_llist *spl_ptr_llist_init(spl_ptr_llist_ctor_func ctor, spl_ptr_llist_dtor_func dtor) /* {{{ */
{
	spl_ptr_llist *llist = emalloc(sizeof(spl_ptr_llist));

	llist->head  = NULL;
	llist->tail  = NULL;
	llist->count = 0;
	llist->dtor  = dtor;
	llist->ctor  = ctor;

	return llist;
}
/* }}} */

static zend_long spl_ptr_llist_count(spl_ptr_llist *llist) /* {{{ */
{
	return (zend_long)llist->count;
}
/* }}} */

static void spl_ptr_llist_destroy(spl_ptr_llist *llist) /* {{{ */
{
	spl_ptr_llist_element   *current = llist->head, *next;
	spl_ptr_llist_dtor_func  dtor    = llist->dtor;

	while (current) {
		next = current->next;
		if (dtor) {
			dtor(current);
		}
		SPL_LLIST_DELREF(current);
		current = next;
	}

	efree(llist);
}
/* }}} */

static spl_ptr_llist_element *spl_ptr_llist_offset(spl_ptr_llist *llist, zend_long offset, int backward) /* {{{ */
{

	spl_ptr_llist_element *current;
	int pos = 0;

	if (backward) {
		current = llist->tail;
	} else {
		current = llist->head;
	}

	while (current && pos < offset) {
		pos++;
		if (backward) {
			current = current->prev;
		} else {
			current = current->next;
		}
	}

	if (current && pos == offset) {
		return current;
	}
	return NULL;
}
/* }}} */

static void spl_ptr_llist_unshift(spl_ptr_llist *llist, zval *data) /* {{{ */
{
	spl_ptr_llist_element *elem = emalloc(sizeof(spl_ptr_llist_element));

	elem->rc   = 1;
	elem->prev = NULL;
	elem->next = llist->head;
	ZVAL_COPY_VALUE(&elem->data, data);

	if (llist->head) {
		llist->head->prev = elem;
	} else {
		llist->tail = elem;
	}

	llist->head = elem;
	llist->count++;

	if (llist->ctor) {
		llist->ctor(elem);
	}
}
/* }}} */

static void spl_ptr_llist_push(spl_ptr_llist *llist, zval *data) /* {{{ */
{
	spl_ptr_llist_element *elem = emalloc(sizeof(spl_ptr_llist_element));

	elem->rc   = 1;
	elem->prev = llist->tail;
	elem->next = NULL;
	ZVAL_COPY_VALUE(&elem->data, data);

	if (llist->tail) {
		llist->tail->next = elem;
	} else {
		llist->head = elem;
	}

	llist->tail = elem;
	llist->count++;

	if (llist->ctor) {
		llist->ctor(elem);
	}
}
/* }}} */

static void spl_ptr_llist_pop(spl_ptr_llist *llist, zval *ret) /* {{{ */
{
	spl_ptr_llist_element    *tail = llist->tail;

	if (tail == NULL) {
		ZVAL_UNDEF(ret);
		return;
	}

	if (tail->prev) {
		tail->prev->next = NULL;
	} else {
		llist->head = NULL;
	}

	llist->tail = tail->prev;
	llist->count--;
	ZVAL_COPY(ret, &tail->data);

	if (llist->dtor) {
		llist->dtor(tail);
	}

	ZVAL_UNDEF(&tail->data);

	SPL_LLIST_DELREF(tail);
}
/* }}} */

static zval *spl_ptr_llist_last(spl_ptr_llist *llist) /* {{{ */
{
	spl_ptr_llist_element *tail = llist->tail;

	if (tail == NULL) {
		return NULL;
	} else {
		return &tail->data;
	}
}
/* }}} */

static zval *spl_ptr_llist_first(spl_ptr_llist *llist) /* {{{ */
{
	spl_ptr_llist_element *head = llist->head;

	if (head == NULL) {
		return NULL;
	} else {
		return &head->data;
	}
}
/* }}} */

static void spl_ptr_llist_shift(spl_ptr_llist *llist, zval *ret) /* {{{ */
{
	spl_ptr_llist_element   *head = llist->head;

	if (head == NULL) {
		ZVAL_UNDEF(ret);
		return;
	}

	if (head->next) {
		head->next->prev = NULL;
	} else {
		llist->tail = NULL;
	}

	llist->head = head->next;
	llist->count--;
	ZVAL_COPY(ret, &head->data);

	if (llist->dtor) {
		llist->dtor(head);
	}
	ZVAL_UNDEF(&head->data);

	SPL_LLIST_DELREF(head);
}
/* }}} */

static void spl_ptr_llist_copy(spl_ptr_llist *from, spl_ptr_llist *to) /* {{{ */
{
	spl_ptr_llist_element *current = from->head, *next;
	//??? spl_ptr_llist_ctor_func ctor = from->ctor;

	while (current) {
		next = current->next;

		/*??? FIXME
		if (ctor) {
			ctor(current);
		}
		*/

		spl_ptr_llist_push(to, &current->data);
		current = next;
	}

}
/* }}} */

/* }}} */

static void spl_dllist_object_free_storage(zend_object *object) /* {{{ */
{
	spl_dllist_object *intern = spl_dllist_from_obj(object);
	zval tmp;

	zend_object_std_dtor(&intern->std);

	while (intern->llist->count > 0) {
		spl_ptr_llist_pop(intern->llist, &tmp);
		zval_ptr_dtor(&tmp);
	}

	if (intern->gc_data != NULL) {
		efree(intern->gc_data);
	};

	spl_ptr_llist_destroy(intern->llist);
	SPL_LLIST_CHECK_DELREF(intern->traverse_pointer);
}
/* }}} */

zend_object_iterator *spl_dllist_get_iterator(zend_class_entry *ce, zval *object, int by_ref);

static zend_object *spl_dllist_object_new_ex(zend_class_entry *class_type, zval *orig, int clone_orig) /* {{{ */
{
	spl_dllist_object *intern;
	zend_class_entry  *parent = class_type;
	int                inherited = 0;

	intern = zend_object_alloc(sizeof(spl_dllist_object), parent);
```

```
377:
378:   zend_object_std_init(&intern->std, class_type);
379:   object_properties_init(&intern->std, class_type);
380:
381:   intern->flags = 0;
382:   intern->traverse_position = 0;
383:
384:   if (orig) {
385:     spl_dllist_object *other = Z_SPLDLLIST_P(orig);
386:     intern->ce_get_iterator = other->ce_get_iterator;
387:
388:     if (clone_orig) {
389:       intern->llist = (spl_ptr_llist *)spl_ptr_llist_init(other->llist->ctor, other->llist->dtor);
390:       spl_ptr_llist_copy(other->llist, intern->llist);
391:       intern->traverse_pointer  = intern->llist->head;
392:       SPL_LLIST_CHECK_ADDREF(intern->traverse_pointer);
393:     } else {
394:       intern->llist = other->llist;
395:       intern->traverse_pointer  = intern->llist->head;
396:       SPL_LLIST_CHECK_ADDREF(intern->traverse_pointer);
397:     }
398:
399:     intern->flags = other->flags;
400:   } else {
401:     intern->llist = (spl_ptr_llist *)spl_ptr_llist_init(spl_ptr_llist_zval_ctor, spl_ptr_llist_zval_dtor);
402:     intern->traverse_pointer  = intern->llist->head;
403:     SPL_LLIST_CHECK_ADDREF(intern->traverse_pointer);
404:   }
405:
406:   while (parent) {
407:     if (parent == spl_ce_SplStack) {
408:       intern->flags |= (SPL_DLLIST_IT_FIX | SPL_DLLIST_IT_LIFO);
409:       intern->std.handlers = &spl_handler_SplDoublyLinkedList;
410:     } else if (parent == spl_ce_SplQueue) {
411:       intern->flags |= SPL_DLLIST_IT_FIX;
412:       intern->std.handlers = &spl_handler_SplDoublyLinkedList;
413:     }
414:
415:     if (parent == spl_ce_SplDoublyLinkedList) {
416:       intern->std.handlers = &spl_handler_SplDoublyLinkedList;
417:       break;
418:     }
419:
420:     parent = parent->parent;
421:     inherited = 1;
422:   }
423:
424:   if (!parent) { /* this must never happen */
425:     php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error, Class is not child of SplDoublyLinkedList");
426:   }
427:   if (inherited) {
428:     intern->fptr_offset_get = zend_hash_str_find_ptr(&class_type->function_table, "offsetget", sizeof("offsetget") - 1);
429:     if (intern->fptr_offset_get->common.scope == parent) {
430:       intern->fptr_offset_get = NULL;
431:     }
432:     intern->fptr_offset_set = zend_hash_str_find_ptr(&class_type->function_table, "offsetset", sizeof("offsetset") - 1);
433:     if (intern->fptr_offset_set->common.scope == parent) {
434:       intern->fptr_offset_set = NULL;
435:     }
436:     intern->fptr_offset_has = zend_hash_str_find_ptr(&class_type->function_table, "offsetexists", sizeof("offsetexists") - 1);
437:     if (intern->fptr_offset_has->common.scope == parent) {
438:       intern->fptr_offset_has = NULL;
439:     }
440:     intern->fptr_offset_del = zend_hash_str_find_ptr(&class_type->function_table, "offsetunset", sizeof("offsetunset") - 1);
441:     if (intern->fptr_offset_del->common.scope == parent) {
442:       intern->fptr_offset_del = NULL;
443:     }
444:     intern->fptr_count = zend_hash_str_find_ptr(&class_type->function_table, "count", sizeof("count") - 1);
445:     if (intern->fptr_count->common.scope == parent) {
446:       intern->fptr_count = NULL;
447:     }
448:   }
449:
450:   return &intern->std;
451: }
452: /* }}} */
453:
454: static zend_object *spl_dllist_object_new(zend_class_entry *class_type) /* {{{ */
455: {
456:   return spl_dllist_object_new_ex(class_type, NULL, 0);
457: }
458: /* }}} */
459:
460: static zend_object *spl_dllist_object_clone(zval *zobject) /* {{{ */
461: {
462:   zend_object        *old_object;
463:   zend_object        *new_object;
464:
465:   old_object  = Z_OBJ_P(zobject);
466:   new_object = spl_dllist_object_new_ex(old_object->ce, zobject, 1);
467:
468:   zend_objects_clone_members(new_object, old_object);
469:
470:   return new_object;
471: }
472: /* }}} */
473:
474: static int spl_dllist_object_count_elements(zval *object, zend_long *count) /* {{{ */
475: {
476:   spl_dllist_object *intern = Z_SPLDLLIST_P(object);
477:
478:   if (intern->fptr_count) {
479:     zval rv;
480:     zend_call_method_with_0_params(object, intern->std.ce, &intern->fptr_count, "count", &rv);
481:     if (!Z_ISUNDEF(rv)) {
482:       *count = zval_get_long(&rv);
483:       zval_ptr_dtor(&rv);
484:       return SUCCESS;
485:     }
486:     *count = 0;
487:     return FAILURE;
488:   }
489:
490:   *count = spl_ptr_llist_count(intern->llist);
491:   return SUCCESS;
492: }
493: /* }}} */
494:
495: static HashTable* spl_dllist_object_get_debug_info(zval *obj, int *is_temp) /* {{{ */
496: {
497:   spl_dllist_object     *intern  = Z_SPLDLLIST_P(obj);
498:   spl_ptr_llist_element *current = intern->llist->head, *next;
499:   zval tmp, dllist_array;
500:   zend_string *pnstr;
501:   int  i = 0;
502:   HashTable *debug_info;
503:   *is_temp = 1;
504:
505:   if (!intern->std.properties) {
506:     rebuild_object_properties(&intern->std);
507:   }
508:
509:   debug_info = zend_new_array(1);
510:   zend_hash_copy(debug_info, intern->std.properties, (copy_ctor_func_t) zval_add_ref);
511:
512:   pnstr = spl_gen_private_prop_name(spl_ce_SplDoublyLinkedList, "flags", sizeof("flags")-1);
513:   ZVAL_LONG(&tmp, intern->flags);
514:   zend_hash_add(debug_info, pnstr, &tmp);
515:   zend_string_release(pnstr);
516:
517:   array_init(&dllist_array);
518:
519:   while (current) {
520:     next = current->next;
521:
522:     add_index_zval(&dllist_array, i, &current->data);
523:     if (Z_REFCOUNTED(current->data)) {
524:       Z_ADDREF(current->data);
525:     }
526:     i++;
527:
528:     current = next;
529:   }
530:
531:   pnstr = spl_gen_private_prop_name(spl_ce_SplDoublyLinkedList, "dllist", sizeof("dllist")-1);
532:   zend_hash_add(debug_info, pnstr, &dllist_array);
533:   zend_string_release(pnstr);
534:
535:   return debug_info;
536: }
537: /* }}} */
538:
539: static HashTable *spl_dllist_object_get_gc(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
540: {
541:   spl_dllist_object *intern  = Z_SPLDLLIST_P(obj);
542:   spl_ptr_llist_element *current = intern->llist->head;
543:   int i = 0;
544:
545:   if (intern->gc_data_count < intern->llist->count) {
546:     intern->gc_data_count = intern->llist->count;
547:     intern->gc_data = safe_erealloc(intern->gc_data, intern->gc_data_count, sizeof(zval), 0);
548:   }
549:
550:   while (current) {
551:     ZVAL_COPY_VALUE(&intern->gc_data[i++], &current->data);
552:     current = current->next;
553:   }
554:
555:   *gc_data = intern->gc_data;
556:   *gc_data_count = i;
557:   return zend_std_get_properties(obj);
558: }
559: /* }}} */
560:
561: /* {{{ proto bool SplDoublyLinkedList::push(mixed value)
562:    Push $value on the SplDoublyLinkedList */
563: SPL_METHOD(SplDoublyLinkedList, push)
564: {
565:   zval *value;
566:   spl_dllist_object *intern;
567:
568:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &value) == FAILURE) {
569:     return;
570:   }
571:
572:   intern = Z_SPLDLLIST_P(getThis());
573:   spl_ptr_llist_push(intern->llist, value);
574:
575:   RETURN_TRUE;
576: }
577: /* }}} */
578:
579: /* {{{ proto bool SplDoublyLinkedList::unshift(mixed value)
580:    Unshift $value on the SplDoublyLinkedList */
581: SPL_METHOD(SplDoublyLinkedList, unshift)
582: {
583:   zval *value;
584:   spl_dllist_object *intern;
585:
586:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &value) == FAILURE) {
587:     return;
588:   }
589:
590:   intern = Z_SPLDLLIST_P(getThis());
591:   spl_ptr_llist_unshift(intern->llist, value);
592:
593:   RETURN_TRUE;
594: }
595: /* }}} */
596:
597: /* {{{ proto mixed SplDoublyLinkedList::pop()
598:    Pop an element out of the SplDoublyLinkedList */
599: SPL_METHOD(SplDoublyLinkedList, pop)
600: {
601:   spl_dllist_object *intern;
602:
603:   if (zend_parse_parameters_none() == FAILURE) {
604:     return;
605:   }
606:
607:   intern = Z_SPLDLLIST_P(getThis());
608:   spl_ptr_llist_pop(intern->llist, return_value);
609:
610:   if (Z_ISUNDEF_P(return_value)) {
611:     zend_throw_exception(spl_ce_RuntimeException, "Can't pop from an empty datastructure", 0);
612:     RETURN_NULL();
613:   }
614: }
615: /* }}} */
616:
617: /* {{{ proto mixed SplDoublyLinkedList::shift()
618:    Shift an element out of the SplDoublyLinkedList */
619: SPL_METHOD(SplDoublyLinkedList, shift)
620: {
621:   spl_dllist_object *intern;
622:
623:   if (zend_parse_parameters_none() == FAILURE) {
624:     return;
625:   }
626:
627:   intern = Z_SPLDLLIST_P(getThis());
628:   spl_ptr_llist_shift(intern->llist, return_value);
629:
630:   if (Z_ISUNDEF_P(return_value)) {
631:     zend_throw_exception(spl_ce_RuntimeException, "Can't shift from an empty datastructure", 0);
632:     RETURN_NULL();
633:   }
634: }
635: /* }}} */
636:
637: /* {{{ proto mixed SplDoublyLinkedList::top()
638:    Peek at the top element of the SplDoublyLinkedList */
639: SPL_METHOD(SplDoublyLinkedList, top)
640: {
641:   zval *value;
642:   spl_dllist_object *intern;
643:
644:   if (zend_parse_parameters_none() == FAILURE) {
645:     return;
646:   }
647:
648:   intern = Z_SPLDLLIST_P(getThis());
649:   value = spl_ptr_llist_last(intern->llist);
650:
651:   if (value == NULL || Z_ISUNDEF_P(value)) {
652:     zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty datastructure", 0);
653:     return;
654:   }
655:
656:   ZVAL_DEREF(value);
657:   ZVAL_COPY(return_value, value);
658: }
659: /* }}} */
660:
661: /* {{{ proto mixed SplDoublyLinkedList::bottom()
662:    Peek at the bottom element of the SplDoublyLinkedList */
663: SPL_METHOD(SplDoublyLinkedList, bottom)
664: {
665:   zval *value;
666:   spl_dllist_object *intern;
667:
668:   if (zend_parse_parameters_none() == FAILURE) {
669:     return;
670:   }
671:
672:   intern = Z_SPLDLLIST_P(getThis());
673:   value  = spl_ptr_llist_first(intern->llist);
674:
675:   if (value == NULL || Z_ISUNDEF_P(value)) {
676:     zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty datastructure", 0);
677:     return;
678:   }
679:
680:   ZVAL_DEREF(value);
681:   ZVAL_COPY(return_value, value);
682: }
683: /* }}} */
684:
685: /* {{{ proto int SplDoublyLinkedList::count()
686:    Return the number of elements in the datastructure. */
687: SPL_METHOD(SplDoublyLinkedList, count)
688: {
689:   zend_long count;
690:   spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());
691:
692:   if (zend_parse_parameters_none() == FAILURE) {
693:     return;
694:   }
695:
696:   count = spl_ptr_llist_count(intern->llist);
697:   RETURN_LONG(count);
698: }
699: /* }}} */
700:
701: /* {{{ proto int SplDoublyLinkedList::isEmpty()
702:    Return true if the SplDoublyLinkedList is empty. */
703: SPL_METHOD(SplDoublyLinkedList, isEmpty)
704: {
705:   zend_long count;
706:
707:   if (zend_parse_parameters_none() == FAILURE) {
708:     return;
709:   }
710:
711:   spl_dllist_object_count_elements(getThis(), &count);
712:   RETURN_BOOL(count == 0);
713: }
714: /* }}} */
715:
716: /* {{{ proto int SplDoublyLinkedList::setIteratorMode(int flags)
717:    Set the mode of iteration */
718: SPL_METHOD(SplDoublyLinkedList, setIteratorMode)
719: {
720:   zend_long value;
721:   spl_dllist_object *intern;
722:
723:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &value) == FAILURE) {
724:     return;
725:   }
726:
727:   intern = Z_SPLDLLIST_P(getThis());
728:
729:   if (intern->flags & SPL_DLLIST_IT_FIX
730:     && (intern->flags & SPL_DLLIST_IT_LIFO) != (value & SPL_DLLIST_IT_LIFO)) {
731:     zend_throw_exception(spl_ce_RuntimeException, "Iterators' LIFO/FIFO modes for SplStack/SplQueue objects are frozen", 0);
732:     return;
733:   }
734:
735:   intern->flags = (value & SPL_DLLIST_IT_MASK) | (intern->flags & SPL_DLLIST_IT_FIX);
736:
737:   RETURN_LONG(intern->flags);
738: }
739: /* }}} */
740:
741: /* {{{ proto int SplDoublyLinkedList::getIteratorMode()
742:    Return the mode of iteration */
743: SPL_METHOD(SplDoublyLinkedList, getIteratorMode)
744: {
745:   spl_dllist_object *intern;
746:
747:   if (zend_parse_parameters_none() == FAILURE) {
748:     return;
749:   }
750:
751:   intern = Z_SPLDLLIST_P(getThis());
752:
```

```
753:   RETURN_LONG(intern->flags);
754: }
755: /* }}} */
756:
757: /* {{{ proto bool SplDoublyLinkedList::offsetExists(mixed index)
758:  Returns whether the requested $index exists. */
759: SPL_METHOD(SplDoublyLinkedList, offsetExists)
760: {
761:   zval             *zindex;
762:   spl_dllist_object *intern;
763:   zend_long                 index;
764:
765:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
766:     return;
767:   }
768:
769:   intern = Z_SPLDLLIST_P(getThis());
770:   index  = spl_offset_convert_to_long(zindex);
771:
772:   RETURN_BOOL(index >= 0 && index < intern->llist->count);
773: } /* }}} */
774:
775: /* {{{ proto mixed SplDoublyLinkedList::offsetGet(mixed index)
776:  Returns the value at the specified $index. */
777: SPL_METHOD(SplDoublyLinkedList, offsetGet)
778: {
779:   zval                  *zindex;
780:   zend_long                     index;
781:   spl_dllist_object     *intern;
782:   spl_ptr_llist_element *element;
783:
784:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
785:     return;
786:   }
787:
788:   intern = Z_SPLDLLIST_P(getThis());
789:   index  = spl_offset_convert_to_long(zindex);
790:
791:   if (index < 0 || index >= intern->llist->count) {
792:     zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid or out of range", 0);
793:     return;
794:   }
795:
796:   element = spl_ptr_llist_offset(intern->llist, index, intern->flags & SPL_DLLIST_IT_LIFO);
797:
798:   if (element != NULL) {
799:     zval *value = &element->data;
800:
801:     ZVAL_DEREF(value);
802:     ZVAL_COPY(return_value, value);
803:   } else {
804:     zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid", 0);
805:   }
806: } /* }}} */
807:
808: /* {{{ proto void SplDoublyLinkedList::offsetSet(mixed index, mixed newval)
809:  Sets the value at the specified $index to $newval. */
810: SPL_METHOD(SplDoublyLinkedList, offsetSet)
811: {
812:   zval                  *zindex, *value;
813:   spl_dllist_object     *intern;
814:
815:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &zindex, &value) == FAILURE) {
816:     return;
817:   }
818:
819:   intern = Z_SPLDLLIST_P(getThis());
820:
821:   if (Z_TYPE_P(zindex) == IS_NULL) {
822:     /* $obj[] = ... */
823:     spl_ptr_llist_push(intern->llist, value);
824:   } else {
825:     /* $obj[$foo] = ... */
826:     zend_long                   index;
827:     spl_ptr_llist_element *element;
828:
829:     index = spl_offset_convert_to_long(zindex);
830:
831:     if (index < 0 || index >= intern->llist->count) {
832:       zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid or out of range", 0);
833:       return;
834:     }
835:
836:     element = spl_ptr_llist_offset(intern->llist, index, intern->flags & SPL_DLLIST_IT_LIFO);
837:
838:     if (element != NULL) {
839:       /* call dtor on the old element as in spl_ptr_llist_pop */
840:       if (intern->llist->dtor) {
841:         intern->llist->dtor(element);
842:       }
843:
844:       /* the element is replaced, delref the old one as in
845:        * SplDoublyLinkedList::pop() */
846:       zval_ptr_dtor(&element->data);
847:       ZVAL_COPY_VALUE(&element->data, value);
848:
849:       /* new element, call ctor as in spl_ptr_llist_push */
850:       if (intern->llist->ctor) {
851:         intern->llist->ctor(element);
852:       }
853:     } else {
854:       zval_ptr_dtor(value);
855:       zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid", 0);
856:       return;
857:     }
858:   }
859: } /* }}} */
860:
861: /* {{{ proto void SplDoublyLinkedList::offsetUnset(mixed index)
862:  Unsets the value at the specified $index. */
863: SPL_METHOD(SplDoublyLinkedList, offsetUnset)
864: {
865:   zval                  *zindex;
866:   zend_long             index;
867:   spl_dllist_object     *intern;
868:   spl_ptr_llist_element *element;
869:   spl_ptr_llist         *llist;
870:
871:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
872:     return;
873:   }
874:
875:   intern = Z_SPLDLLIST_P(getThis());
876:   index  = spl_offset_convert_to_long(zindex);
877:   llist  = intern->llist;
878:
879:   if (index < 0 || index >= intern->llist->count) {
880:     zend_throw_exception(spl_ce_OutOfRangeException, "Offset out of range", 0);
881:     return;
882:   }
883:
884:   element = spl_ptr_llist_offset(intern->llist, index, intern->flags & SPL_DLLIST_IT_LIFO);
885:
886:   if (element != NULL) {
887:     /* connect the neighbors */
888:     if (element->prev) {
889:       element->prev->next = element->next;
890:     }
891:
892:     if (element->next) {
893:       element->next->prev = element->prev;
894:     }
895:
896:     /* take care of head/tail */
897:     if (element == llist->head) {
898:       llist->head = element->next;
899:     }
900:
901:     if (element == llist->tail) {
902:       llist->tail = element->prev;
903:     }
904:
905:     /* finally, delete the element */
906:     llist->count--;
907:
908:     if(llist->dtor) {
909:       llist->dtor(element);
910:     }
911:
912:     if (intern->traverse_pointer == element) {
913:       SPL_LLIST_DELREF(element);
914:       intern->traverse_pointer = NULL;
915:     }
916:     zval_ptr_dtor(&element->data);
917:     ZVAL_UNDEF(&element->data);
918:
919:     SPL_LLIST_DELREF(element);
920:   } else {
921:     zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid", 0);
922:     return;
923:   }
924: } /* }}} */
925:
926: static void spl_dllist_it_dtor(zend_object_iterator *iter) /* {{{ */
927: {
928:   spl_dllist_it *iterator = (spl_dllist_it *)iter;
929:
930:   SPL_LLIST_CHECK_DELREF(iterator->traverse_pointer);
931:
932:   zend_user_it_invalidate_current(iter);
933:   zval_ptr_dtor(&iterator->intern.it.data);
934: }
935: /* }}} */
936:
937: static void spl_dllist_it_helper_rewind(spl_ptr_llist_element **traverse_pointer_ptr, int *traverse_position_ptr, spl_ptr_llist *llist, int flags) /* {
{{ */
938: {
939:   SPL_LLIST_CHECK_DELREF(*traverse_pointer_ptr);
940:
941:   if (flags & SPL_DLLIST_IT_LIFO) {
942:     *traverse_position_ptr = llist->count-1;
943:     *traverse_pointer_ptr  = llist->tail;
944:   } else {
945:     *traverse_position_ptr = 0;
946:     *traverse_pointer_ptr  = llist->head;
947:   }
948:
949:   SPL_LLIST_CHECK_ADDREF(*traverse_pointer_ptr);
950: }
951: /* }}} */
952:
953: static void spl_dllist_it_helper_move_forward(spl_ptr_llist_element **traverse_pointer_ptr, int *traverse_position_ptr, spl_ptr_llist *llist, int flags
) /* {{{ */
954: {
955:   if (*traverse_pointer_ptr) {
956:     spl_ptr_llist_element *old = *traverse_pointer_ptr;
957:
958:     if (flags & SPL_DLLIST_IT_LIFO) {
959:       *traverse_pointer_ptr = old->prev;
960:       (*traverse_position_ptr)--;
961:
962:       if (flags & SPL_DLLIST_IT_DELETE) {
963:         zval prev;
964:         spl_ptr_llist_pop(llist, &prev);
965:
966:         zval_ptr_dtor(&prev);
967:       }
968:     } else {
969:       *traverse_pointer_ptr = old->next;
970:
971:       if (flags & SPL_DLLIST_IT_DELETE) {
972:         zval prev;
973:         spl_ptr_llist_shift(llist, &prev);
974:
975:         zval_ptr_dtor(&prev);
976:       } else {
977:         (*traverse_position_ptr)++;
978:       }
979:     }
980:
981:     SPL_LLIST_DELREF(old);
982:     SPL_LLIST_CHECK_ADDREF(*traverse_pointer_ptr);
983:   }
984: }
985: /* }}} */
986:
987: static void spl_dllist_it_rewind(zend_object_iterator *iter) /* {{{ */
988: {
989:   spl_dllist_it *iterator = (spl_dllist_it *)iter;
990:   spl_dllist_object *object = Z_SPLDLLIST_P(&iter->data);
991:   spl_ptr_llist *llist = object->llist;
992:
993:   spl_dllist_it_helper_rewind(&iterator->traverse_pointer, &iterator->traverse_position, llist, object->flags);
994: }
995: /* }}} */
996:
997: static int spl_dllist_it_valid(zend_object_iterator *iter) /* {{{ */
998: {
999:   spl_dllist_it         *iterator = (spl_dllist_it *)iter;
1000:   spl_ptr_llist_element *element  = iterator->traverse_pointer;
1001:
1002:   return (element != NULL ? SUCCESS : FAILURE);
1003: }
1004: /* }}} */
1005:
1006: static zval *spl_dllist_it_get_current_data(zend_object_iterator *iter) /* {{{ */
1007: {
1008:   spl_dllist_it         *iterator = (spl_dllist_it *)iter;
1009:   spl_ptr_llist_element *element  = iterator->traverse_pointer;
1010:
1011:   if (element == NULL || Z_ISUNDEF(element->data)) {
1012:     return NULL;
1013:   }
1014:
1015:   return &element->data;
1016: }
1017: /* }}} */
1018:
1019: static void spl_dllist_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
1020: {
1021:   spl_dllist_it *iterator = (spl_dllist_it *)iter;
1022:
1023:   ZVAL_LONG(key, iterator->traverse_position);
1024: }
1025: /* }}} */
1026:
1027: static void spl_dllist_it_move_forward(zend_object_iterator *iter) /* {{{ */
1028: {
1029:   spl_dllist_it *iterator = (spl_dllist_it *)iter;
1030:   spl_dllist_object *object = Z_SPLDLLIST_P(&iter->data);
1031:
1032:   zend_user_it_invalidate_current(iter);
1033:
1034:   spl_dllist_it_helper_move_forward(&iterator->traverse_pointer, &iterator->traverse_position, object->llist, object->flags);
1035: }
1036: /* }}} */
1037:
1038: /* {{{  proto int SplDoublyLinkedList::key()
1039:    Return current array key */
1040: SPL_METHOD(SplDoublyLinkedList, key)
1041: {
1042:   spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());
1043:
1044:   if (zend_parse_parameters_none() == FAILURE) {
1045:     return;
1046:   }
1047:
1048:   RETURN_LONG(intern->traverse_position);
1049: }
1050: /* }}} */
1051:
1052: /* {{{ proto void SplDoublyLinkedList::prev()
1053:    Move to next entry */
1054: SPL_METHOD(SplDoublyLinkedList, prev)
1055: {
1056:   spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());
1057:
1058:   if (zend_parse_parameters_none() == FAILURE) {
1059:     return;
1060:   }
1061:
1062:   spl_dllist_it_helper_move_forward(&intern->traverse_pointer, &intern->traverse_position, intern->llist, intern->flags ^ SPL_DLLIST_IT_LIFO);
1063: }
1064: /* }}} */
1065:
1066: /* {{{ proto void SplDoublyLinkedList::next()
1067:    Move to next entry */
1068: SPL_METHOD(SplDoublyLinkedList, next)
1069: {
1070:   spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());
1071:
1072:   if (zend_parse_parameters_none() == FAILURE) {
1073:     return;
1074:   }
1075:
1076:   spl_dllist_it_helper_move_forward(&intern->traverse_pointer, &intern->traverse_position, intern->llist, intern->flags);
1077: }
1078: /* }}} */
1079:
1080: /* {{{ proto bool SplDoublyLinkedList::valid()
1081:    Check whether the datastructure contains more entries */
1082: SPL_METHOD(SplDoublyLinkedList, valid)
1083: {
1084:   spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());
1085:
1086:   if (zend_parse_parameters_none() == FAILURE) {
1087:     return;
1088:   }
1089:
1090:   RETURN_BOOL(intern->traverse_pointer != NULL);
1091: }
1092: /* }}} */
1093:
1094: /* {{{ proto void SplDoublyLinkedList::rewind()
1095:    Rewind the datastructure back to the start */
1096: SPL_METHOD(SplDoublyLinkedList, rewind)
1097: {
1098:   spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());
1099:
1100:   if (zend_parse_parameters_none() == FAILURE) {
1101:     return;
1102:   }
1103:
1104:   spl_dllist_it_helper_rewind(&intern->traverse_pointer, &intern->traverse_position, intern->llist, intern->flags);
1105: }
1106: /* }}} */
1107:
1108: /* {{{ proto mixed|NULL SplDoublyLinkedList::current()
1109:    Return current datastructure entry */
1110: SPL_METHOD(SplDoublyLinkedList, current)
1111: {
1112:   spl_dllist_object     *intern  = Z_SPLDLLIST_P(getThis());
1113:   spl_ptr_llist_element *element = intern->traverse_pointer;
1114:
1115:   if (zend_parse_parameters_none() == FAILURE) {
1116:     return;
1117:   }
1118:
1119:   if (element == NULL || Z_ISUNDEF(element->data)) {
1120:     RETURN_NULL();
1121:   } else {
1122:     zval *value = &element->data;
1123:
1124:     ZVAL_DEREF(value);
1125:     ZVAL_COPY(return_value, value);
1126:   }
```

```
} /* }}} */

/* {{{ proto string SplDoublyLinkedList::serialize()
   Serializes storage */
SPL_METHOD(SplDoublyLinkedList, serialize)
{
	spl_dllist_object     *intern   = Z_SPLDLLIST_P(getThis());
	smart_str              buf      = {0};
	spl_ptr_llist_element *current  = intern->llist->head, *next;
	zval                   flags;
	php_serialize_data_t   var_hash;

	if (zend_parse_parameters_none() == FAILURE) {
		return;
	}

	PHP_VAR_SERIALIZE_INIT(var_hash);

	/* flags */
	ZVAL_LONG(&flags, intern->flags);
	php_var_serialize(&buf, &flags, &var_hash);
	zval_ptr_dtor(&flags);

	/* elements */
	while (current) {
		smart_str_appendc(&buf, ':');
		next = current->next;

		php_var_serialize(&buf, &current->data, &var_hash);

		current = next;
	}

	smart_str_0(&buf);

	/* done */
	PHP_VAR_SERIALIZE_DESTROY(var_hash);

	if (buf.s) {
		RETURN_NEW_STR(buf.s);
	} else {
		RETURN_NULL();
	}
} /* }}} */

/* {{{ proto void SplDoublyLinkedList::unserialize(string serialized)
   Unserializes storage */
SPL_METHOD(SplDoublyLinkedList, unserialize)
{
	spl_dllist_object *intern = Z_SPLDLLIST_P(getThis());
	zval *flags, *elem;
	char *buf;
	size_t buf_len;
	const unsigned char *p, *s;
	php_unserialize_data_t var_hash;

	if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &buf, &buf_len) == FAILURE) {
		return;
	}

	if (buf_len == 0) {
		return;
	}

	s = p = (const unsigned char*)buf;
	PHP_VAR_UNSERIALIZE_INIT(var_hash);

	/* flags */
	flags = var_tmp_var(&var_hash);
	if (!php_var_unserialize(flags, &p, s + buf_len, &var_hash) || Z_TYPE_P(flags) != IS_LONG) {
		goto error;
	}

	intern->flags = (int)Z_LVAL_P(flags);

	/* elements */
	while(*p == ':') {
		++p;
		elem = var_tmp_var(&var_hash);
		if (!php_var_unserialize(elem, &p, s + buf_len, &var_hash)) {
			goto error;
		}
		var_push_dtor(&var_hash, elem);

		spl_ptr_llist_push(intern->llist, elem);
	}

	if (*p != '\0') {
		goto error;
	}

	PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
	return;

error:
	PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
	zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Error at offset %zd of %zd bytes", ((char*)p - buf), buf_len);
	return;

} /* }}} */

/* {{{ proto void SplDoublyLinkedList::add(mixed index, mixed newval)
   Inserts a new entry before the specified $index consisting of $newval. */
SPL_METHOD(SplDoublyLinkedList, add)
{
	zval                  *zindex, *value;
	spl_dllist_object     *intern;
	spl_ptr_llist_element *element;
	zend_long              index;

	if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &zindex, &value) == FAILURE) {
		return;
	}

	intern = Z_SPLDLLIST_P(getThis());
	index  = spl_offset_convert_to_long(zindex);

	if (index < 0 || index > intern->llist->count) {
		zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid or out of range", 0);
		return;
	}

	Z_TRY_ADDREF_P(value);
	if (index == intern->llist->count) {
		/* If index is the last entry+1 then we do a push because we're not inserting before any entry */
		spl_ptr_llist_push(intern->llist, value);
	} else {
		/* Create the new element we want to insert */
		spl_ptr_llist_element *elem = emalloc(sizeof(spl_ptr_llist_element));

		/* Get the element we want to insert before */
		element = spl_ptr_llist_offset(intern->llist, index, intern->flags & SPL_DLLIST_IT_LIFO);

		ZVAL_COPY_VALUE(&elem->data, value);
		elem->rc   = 1;
		/* connect to the neighbours */
		elem->next = element;
		elem->prev = element->prev;

		/* connect the neighbours to this new element */
		if (elem->prev == NULL) {
			intern->llist->head = elem;
		} else {
			element->prev->next = elem;
		}
		element->prev = elem;

		intern->llist->count++;

		if (intern->llist->ctor) {
			intern->llist->ctor(elem);
		}
	}
} /* }}} */

/* {{{ iterator handler table */
static const zend_object_iterator_funcs spl_dllist_it_funcs = {
	spl_dllist_it_dtor,
	spl_dllist_it_valid,
	spl_dllist_it_get_current_data,
	spl_dllist_it_get_current_key,
	spl_dllist_it_move_forward,
	spl_dllist_it_rewind,
	NULL
}; /* }}} */

zend_object_iterator *spl_dllist_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
{
	spl_dllist_it *iterator;
	spl_dllist_object *dllist_object = Z_SPLDLLIST_P(object);

	if (by_ref) {
		zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
		return NULL;
	}

	iterator = emalloc(sizeof(spl_dllist_it));

	zend_iterator_init((zend_object_iterator*)iterator);

	ZVAL_COPY(&iterator->intern.it.data, object);
	iterator->intern.it.funcs    = &spl_dllist_it_funcs;
	iterator->intern.ce          = ce;
	iterator->traverse_position  = dllist_object->traverse_position;
	iterator->traverse_pointer   = dllist_object->traverse_pointer;
	iterator->flags              = dllist_object->flags & SPL_DLLIST_IT_MASK;
	ZVAL_UNDEF(&iterator->intern.value);

	SPL_LLIST_CHECK_ADDREF(iterator->traverse_pointer);

	return &iterator->intern.it;
}
/* }}} */

/*  Function/Class/Method definitions */
ZEND_BEGIN_ARG_INFO(arginfo_dllist_setiteratormode, 0)
	ZEND_ARG_INFO(0, flags)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_dllist_push, 0)
	ZEND_ARG_INFO(0, value)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_dllist_offsetGet, 0, 0, 1)
	ZEND_ARG_INFO(0, index)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO_EX(arginfo_dllist_offsetSet, 0, 0, 2)
	ZEND_ARG_INFO(0, index)
	ZEND_ARG_INFO(0, newval)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_dllist_void, 0)
ZEND_END_ARG_INFO()

ZEND_BEGIN_ARG_INFO(arginfo_dllist_serialized, 0)
	ZEND_ARG_INFO(0, serialized)
ZEND_END_ARG_INFO();

static const zend_function_entry spl_funcs_SplQueue[] = {
	SPL_MA(SplQueue, enqueue, SplDoublyLinkedList, push,  arginfo_dllist_push, ZEND_ACC_PUBLIC)
	SPL_MA(SplQueue, dequeue, SplDoublyLinkedList, shift, arginfo_dllist_void, ZEND_ACC_PUBLIC)
	PHP_FE_END
};

static const zend_function_entry spl_funcs_SplDoublyLinkedList[] = {
	SPL_ME(SplDoublyLinkedList, pop,             arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, shift,           arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, push,            arginfo_dllist_push,            ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, unshift,         arginfo_dllist_push,            ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, top,             arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, bottom,          arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, isEmpty,         arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, setIteratorMode, arginfo_dllist_setiteratormode, ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, getIteratorMode, arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	/* Countable */
	SPL_ME(SplDoublyLinkedList, count,           arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	/* ArrayAccess */
	SPL_ME(SplDoublyLinkedList, offsetExists,    arginfo_dllist_offsetGet,       ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, offsetGet,       arginfo_dllist_offsetGet,       ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, offsetSet,       arginfo_dllist_offsetSet,       ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, offsetUnset,     arginfo_dllist_offsetGet,       ZEND_ACC_PUBLIC)

	SPL_ME(SplDoublyLinkedList, add,             arginfo_dllist_offsetSet,       ZEND_ACC_PUBLIC)

	/* Iterator */
	SPL_ME(SplDoublyLinkedList, rewind,          arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, current,         arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, key,             arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, next,            arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, prev,            arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList, valid,           arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	/* Serializable */
	SPL_ME(SplDoublyLinkedList,  unserialize,    arginfo_dllist_serialized,      ZEND_ACC_PUBLIC)
	SPL_ME(SplDoublyLinkedList,  serialize,      arginfo_dllist_void,            ZEND_ACC_PUBLIC)
	PHP_FE_END
};
/* }}} */

PHP_MINIT_FUNCTION(spl_dllist) /* {{{ */
{
	REGISTER_SPL_STD_CLASS_EX(SplDoublyLinkedList, spl_dllist_object_new, spl_funcs_SplDoublyLinkedList);
	memcpy(&spl_handler_SplDoublyLinkedList, zend_get_std_object_handlers(), sizeof(zend_object_handlers));

	spl_handler_SplDoublyLinkedList.offset = XtOffsetOf(spl_dllist_object, std);
	spl_handler_SplDoublyLinkedList.clone_obj = spl_dllist_object_clone;
	spl_handler_SplDoublyLinkedList.count_elements = spl_dllist_object_count_elements;
	spl_handler_SplDoublyLinkedList.get_debug_info = spl_dllist_object_get_debug_info;
	spl_handler_SplDoublyLinkedList.get_gc = spl_dllist_object_get_gc;
	spl_handler_SplDoublyLinkedList.dtor_obj = zend_objects_destroy_object;
	spl_handler_SplDoublyLinkedList.free_obj = spl_dllist_object_free_storage;

	REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_LIFO",  SPL_DLLIST_IT_LIFO);
	REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_FIFO",  0);
	REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_DELETE",SPL_DLLIST_IT_DELETE);
	REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_KEEP",  0);

	REGISTER_SPL_IMPLEMENTS(SplDoublyLinkedList, Iterator);
	REGISTER_SPL_IMPLEMENTS(SplDoublyLinkedList, Countable);
	REGISTER_SPL_IMPLEMENTS(SplDoublyLinkedList, ArrayAccess);
	REGISTER_SPL_IMPLEMENTS(SplDoublyLinkedList, Serializable);

	spl_ce_SplDoublyLinkedList->get_iterator = spl_dllist_get_iterator;

	REGISTER_SPL_SUB_CLASS_EX(SplQueue,          SplDoublyLinkedList,       spl_dllist_object_new, spl_funcs_SplQueue);
	REGISTER_SPL_SUB_CLASS_EX(SplStack,          SplDoublyLinkedList,       spl_dllist_object_new, NULL);

	spl_ce_SplQueue->get_iterator = spl_dllist_get_iterator;
	spl_ce_SplStack->get_iterator = spl_dllist_get_iterator;

	return SUCCESS;
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "zend_interfaces.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"

#include "spl_array.h"

/* {{{ spl_instantiate */
PHPAPI void spl_instantiate(zend_class_entry *pce, zval *object)
{
	object_init_ex(object, pce);
}
/* }}} */

PHPAPI zend_long spl_offset_convert_to_long(zval *offset) /* {{{ */
{
	zend_ulong idx;

try_again:
	switch (Z_TYPE_P(offset)) {
	case IS_STRING:
		if (ZEND_HANDLE_NUMERIC(Z_STR_P(offset), idx)) {
			return idx;
		}
		break;
	case IS_DOUBLE:
		return (zend_long)Z_DVAL_P(offset);
	case IS_LONG:
		return Z_LVAL_P(offset);
	case IS_FALSE:
		return 0;
	case IS_TRUE:
		return 1;
	case IS_REFERENCE:
		offset = Z_REFVAL_P(offset);
		goto try_again;
	case IS_RESOURCE:
		return Z_RES_HANDLE_P(offset);
	}
	return -1;
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Author: Marcus Boerger <helly@php.net>                               |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "ext/standard/file.h"
#include "ext/standard/php_string.h"
#include "zend_compile.h"
#include "zend_exceptions.h"
#include "zend_interfaces.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_iterators.h"
#include "spl_directory.h"
#include "spl_exceptions.h"

#include "php.h"
#include "fopen_wrappers.h"

#include "ext/standard/basic_functions.h"
#include "ext/standard/php_filestat.h"

#define SPL_HAS_FLAG(flags, test_flag) ((flags & test_flag) ? 1 : 0)

/* declare the class handlers */
static zend_object_handlers spl_filesystem_object_handlers;
/* includes handler to validate object state when retrieving methods */
static zend_object_handlers spl_filesystem_object_check_handlers;

/* decalre the class entry */
PHPAPI zend_class_entry *spl_ce_SplFileInfo;
PHPAPI zend_class_entry *spl_ce_DirectoryIterator;
PHPAPI zend_class_entry *spl_ce_FilesystemIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveDirectoryIterator;
PHPAPI zend_class_entry *spl_ce_GlobIterator;
PHPAPI zend_class_entry *spl_ce_SplFileObject;
PHPAPI zend_class_entry *spl_ce_SplTempFileObject;

static void spl_filesystem_file_free_line(spl_filesystem_object *intern) /* {{{ */
{
	if (intern->u.file.current_line) {
		efree(intern->u.file.current_line);
		intern->u.file.current_line = NULL;
	}
	if (!Z_ISUNDEF(intern->u.file.current_zval)) {
		zval_ptr_dtor(&intern->u.file.current_zval);
		ZVAL_UNDEF(&intern->u.file.current_zval);
	}
} /* }}} */

static void spl_filesystem_object_destroy_object(zend_object *object) /* {{{ */
{
	spl_filesystem_object *intern = spl_filesystem_from_obj(object);

	zend_objects_destroy_object(object);

	switch(intern->type) {
	case SPL_FS_DIR:
		if (intern->u.dir.dirp) {
			php_stream_close(intern->u.dir.dirp);
			intern->u.dir.dirp = NULL;
		}
		break;
	case SPL_FS_FILE:
		if (intern->u.file.stream) {
			/*
			if (intern->u.file.zcontext) {
				zend_list_delref(Z_RESVAL_P(intern->zcontext));
			}
			*/
			if (!intern->u.file.stream->is_persistent) {
				php_stream_close(intern->u.file.stream);
			} else {
				php_stream_pclose(intern->u.file.stream);
			}
		}
		break;
	default:
		break;
	}
} /* }}} */

static void spl_filesystem_object_free_storage(zend_object *object) /* {{{ */
{
	spl_filesystem_object *intern = spl_filesystem_from_obj(object);

	if (intern->oth_handler && intern->oth_handler->dtor) {
		intern->oth_handler->dtor(intern);
	}

	zend_object_std_dtor(&intern->std);

	if (intern->_path) {
		efree(intern->_path);
	}
	if (intern->file_name) {
		efree(intern->file_name);
	}
	switch(intern->type) {
	case SPL_FS_INFO:
		break;
	case SPL_FS_DIR:
		if (intern->u.dir.sub_path) {
			efree(intern->u.dir.sub_path);
		}
		break;
	case SPL_FS_FILE:
		if (intern->u.file.stream) {
			if (intern->u.file.open_mode) {
				efree(intern->u.file.open_mode);
			}
			if (intern->orig_path) {
				efree(intern->orig_path);
			}
		}
		spl_filesystem_file_free_line(intern);
		break;
	}
} /* }}} */

/* {{{ spl_ce_dir_object_new */
/* creates the object by
   - allocating memory
   - initializing the object members
   - storing the object
   - setting it's handlers

   called from
   - clone
   - new
 */
static zend_object *spl_filesystem_object_new_ex(zend_class_entry *class_type)
{
	spl_filesystem_object *intern;

	intern = zend_object_alloc(sizeof(spl_filesystem_object), class_type);
	/* intern->type = SPL_FS_INFO; done by set 0 */
	intern->file_class = spl_ce_SplFileObject;
	intern->info_class = spl_ce_SplFileInfo;

	zend_object_std_init(&intern->std, class_type);
	object_properties_init(&intern->std, class_type);
	intern->std.handlers = &spl_filesystem_object_handlers;

	return &intern->std;
}
/* }}} */

/* {{{ spl_filesystem_object_new */
/* See spl_filesystem_object_new_ex */
static zend_object *spl_filesystem_object_new(zend_class_entry *class_type)
{
	return spl_filesystem_object_new_ex(class_type);
}
/* }}} */

/* {{{ spl_filesystem_object_new_check */
static zend_object *spl_filesystem_object_new_check(zend_class_entry *class_type)
{
	spl_filesystem_object *ret = spl_filesystem_from_obj(spl_filesystem_object_new_ex(class_type));
	ret->std.handlers = &spl_filesystem_object_check_handlers;
	return &ret->std;
}
/* }}} */

PHPAPI char* spl_filesystem_object_get_path(spl_filesystem_object *intern, size_t *len) /* {{{ */
{
#ifdef HAVE_GLOB
	if (intern->type == SPL_FS_DIR) {
		if (php_stream_is(intern->u.dir.dirp ,&php_glob_stream_ops)) {
			return php_glob_stream_get_path(intern->u.dir.dirp, 0, len);
		}
	}
#endif
	if (len) {
		*len = intern->_path_len;
	}
	return intern->_path;
} /* }}} */

static inline void spl_filesystem_object_get_file_name(spl_filesystem_object *intern) /* {{{ */
{
	char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIXPATHS) ? '/' : DEFAULT_SLASH;

	switch (intern->type) {
		case SPL_FS_INFO:
		case SPL_FS_FILE:
			if (!intern->file_name) {
				php_error_docref(NULL, E_ERROR, "Object not initialized");
			}
			break;
		case SPL_FS_DIR:
			if (intern->file_name) {
				efree(intern->file_name);
			}
			intern->file_name_len = spprintf(&intern->file_name, 0, "%s%c%s",
			                                 spl_filesystem_object_get_path(intern, NULL),
			                                 slash, intern->u.dir.entry.d_name);
			break;
	}
} /* }}} */

static int spl_filesystem_dir_read(spl_filesystem_object *intern) /* {{{ */
{
	if (!intern->u.dir.dirp || !php_stream_readdir(intern->u.dir.dirp, &intern->u.dir.entry)) {
		intern->u.dir.entry.d_name[0] = '\0';
		return 0;
	} else {
		return 1;
	}
}
/* }}} */

#define IS_SLASH_AT(zs, pos) (IS_SLASH(zs[pos]))

static inline int spl_filesystem_is_dot(const char * d_name) /* {{{ */
{
	return !strcmp(d_name, ".") || !strcmp(d_name, "..");
}
/* }}} */

/* {{{ spl_filesystem_dir_open */
/* open a directory resource */
static void spl_filesystem_dir_open(spl_filesystem_object* intern, char *path)
{
	int skip_dots = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_SKIPDOTS);

	intern->type = SPL_FS_DIR;
	intern->_path_len = strlen(path);
	intern->u.dir.dirp = php_stream_opendir(path, REPORT_ERRORS, FG(default_context));

	if (intern->_path_len > 1 && IS_SLASH_AT(path, intern->_path_len-1)) {
		intern->_path = estrndup(path, --intern->_path_len);
	} else {
		intern->_path = estrndup(path, intern->_path_len);
	}
	intern->u.dir.index = 0;

	if (EG(exception) || intern->u.dir.dirp == NULL) {
		intern->u.dir.entry.d_name[0] = '\0';
		if (!EG(exception)) {
			/* open failed w/out notice (turned to exception due to EH_THROW) */
			zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0,
				"Failed to open directory \"%s\"", path);
		}
	} else {
		do {
			spl_filesystem_dir_read(intern);
		} while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry.d_name));
	}
}
/* }}} */

/* {{{ spl_filesystem_file_open */
static int spl_filesystem_file_open(spl_filesystem_object *intern, int use_include_path, int silent) /* {{{ */
{
	zval tmp;

	intern->type = SPL_FS_FILE;

	php_stat(intern->file_name, intern->file_name_len, FS_IS_DIR, &tmp);
	if (Z_TYPE(tmp) == IS_TRUE) {
		intern->u.file.open_mode = NULL;
		intern->file_name = NULL;
		zend_throw_exception_ex(spl_ce_LogicException, 0, "Cannot use SplFileObject with directories");
		return FAILURE;
	}

	intern->u.file.context = php_stream_context_from_zval(intern->u.file.zcontext, 0);
	intern->u.file.stream = php_stream_open_wrapper_ex(intern->file_name, intern->u.file.open_mode, (use_include_path ? USE_PATH : 0) | REPORT_ERRORS, NULL, intern->u.file.context);

	if (!intern->file_name_len || !intern->u.file.stream) {
		if (!EG(exception)) {
			zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot open file '%s'", intern->file_name_len ? intern->file_name : "");
		}
		intern->file_name = NULL; /* until here it is not a copy */
		intern->u.file.open_mode = NULL;
		return FAILURE;
	}

	/*
	if (intern->u.file.zcontext) {
		//zend_list_addref(Z_RES_VAL(intern->u.file.zcontext));
		Z_ADDREF_P(intern->u.file.zcontext);
	}
	*/

	if (intern->file_name_len > 1 && IS_SLASH_AT(intern->file_name, intern->file_name_len-1)) {
		intern->file_name_len--;
	}

	intern->orig_path = estrndup(intern->u.file.stream->orig_path, strlen(intern->u.file.stream->orig_path));

	intern->file_name = estrndup(intern->file_name, intern->file_name_len);
	intern->u.file.open_mode = estrndup(intern->u.file.open_mode, intern->u.file.open_mode_len);

	/* avoid reference counting in debug mode, thus do it manually */
	ZVAL_RES(&intern->u.file.zresource, intern->u.file.stream->res);
	/*!!! TODO: maybe bug?
	Z_SET_REFCOUNT(intern->u.file.zresource, 1);
	*/

	intern->u.file.delimiter = ',';
	intern->u.file.enclosure = '"';
	intern->u.file.escape = '\\';

	intern->u.file.func_getCurr = zend_hash_str_find_ptr(&intern->std.ce->function_table, "getcurrentline", sizeof("getcurrentline") - 1);

	return SUCCESS;
} /* }}} */

/* {{{ spl_filesystem_object_clone */
/* Local zend_object creation (on stack)
   Load the 'other' object
   Create a new empty object (See spl_filesystem_object_new_ex)
   Open the directory
   Clone other members (properties)
 */
static zend_object *spl_filesystem_object_clone(zval *zobject)
{
	zend_object *old_object;
	zend_object *new_object;
	spl_filesystem_object *intern;
	spl_filesystem_object *source;
	int index, skip_dots;

	old_object = Z_OBJ_P(zobject);
	source = spl_filesystem_from_obj(old_object);
	new_object = spl_filesystem_object_new_ex(old_object->ce);
	intern = spl_filesystem_from_obj(new_object);

	intern->flags = source->flags;

	switch (source->type) {
		case SPL_FS_INFO:
			intern->_path_len = source->_path_len;
			intern->_path = estrndup(source->_path, source->_path_len);
			intern->file_name_len = source->file_name_len;
			intern->file_name = estrndup(source->file_name, intern->file_name_len);
			break;
		case SPL_FS_DIR:
			spl_filesystem_dir_open(intern, source->_path);
			/* read until we hit the position in which we were before */
			skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
			for(index = 0; index < source->u.dir.index; ++index) {
				do {
					spl_filesystem_dir_read(intern);
				} while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry.d_name));
			}
			intern->u.dir.index = index;
			break;
		case SPL_FS_FILE:
```

```
376:       zend_throw_error(NULL, "An object of class %s cannot be cloned", ZSTR_VAL(old_object->ce->name));
377:       return new_object;
378:   }
379:
380:   intern->file_class = source->file_class;
381:   intern->info_class = source->info_class;
382:   intern->oth = source->oth;
383:   intern->oth_handler = source->oth_handler;
384:
385:   zend_objects_clone_members(new_object, old_object);
386:
387:   if (intern->oth_handler && intern->oth_handler->clone) {
388:     intern->oth_handler->clone(source, intern);
389:   }
390:
391:   return new_object;
392: }
393: /* }}} */
394:
395: void spl_filesystem_info_set_filename(spl_filesystem_object *intern, char *path, size_t len, size_t use_copy) /* {{{ */
396: {
397:   char *p1, *p2;
398:
399:   if (intern->file_name) {
400:     efree(intern->file_name);
401:   }
402:
403:   intern->file_name = use_copy ? estrndup(path, len) : path;
404:   intern->file_name_len = len;
405:
406:   while (intern->file_name_len > 1 && IS_SLASH_AT(intern->file_name, intern->file_name_len-1)) {
407:     intern->file_name[intern->file_name_len-1] = 0;
408:     intern->file_name_len--;
409:   }
410:
411:   p1 = strrchr(intern->file_name, '/');
412: #if defined(PHP_WIN32)
413:   p2 = strrchr(intern->file_name, '\\');
414: #else
415:   p2 = 0;
416: #endif
417:   if (p1 || p2) {
418:     intern->_path_len = ((p1 > p2 ? p1 : p2) - intern->file_name);
419:   } else {
420:     intern->_path_len = 0;
421:   }
422:
423:   if (intern->_path) {
424:     efree(intern->_path);
425:   }
426:   intern->_path = estrndup(path, intern->_path_len);
427: } /* }}} */
428:
429: static spl_filesystem_object *spl_filesystem_object_create_info(spl_filesystem_object *source, char *file_path, size_t file_path_len, int use_copy, zend_class_entry *ce, zval *return_value) /* {{{ */
430: {
431:   spl_filesystem_object *intern;
432:   zval arg1;
433:   zend_error_handling error_handling;
434:
435:   if (!file_path || !file_path_len) {
436: #if defined(PHP_WIN32)
437:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot create SplFileInfo for empty path");
438:     if (file_path && !use_copy) {
439:       efree(file_path);
440:     }
441: #else
442:     if (file_path && !use_copy) {
443:       efree(file_path);
444:     }
445:     file_path_len = 1;
446:     file_path = "/";
447: #endif
448:     return NULL;
449:   }
450:
451:   zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
452:
453:   ce = ce ? ce : source->info_class;
454:
455:   zend_update_class_constants(ce);
456:
457:   intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
458:   ZVAL_OBJ(return_value, &intern->std);
459:
460:   if (ce->constructor->common.scope != spl_ce_SplFileInfo) {
461:     ZVAL_STRINGL(&arg1, file_path, file_path_len);
462:     zend_call_method_with_1_params(return_value, ce, &ce->constructor, "__construct", NULL, &arg1);
463:     zval_ptr_dtor(&arg1);
464:   } else {
465:     spl_filesystem_info_set_filename(intern, file_path, file_path_len, use_copy);
466:   }
467:
468:   zend_restore_error_handling(&error_handling);
469:   return intern;
470: } /* }}} */
471:
472: static spl_filesystem_object *spl_filesystem_object_create_type(int ht, spl_filesystem_object *source, int type, zend_class_entry *ce, zval *return_value) /* {{{ */
473: {
474:   spl_filesystem_object *intern;
475:   zend_bool use_include_path = 0;
476:   zval arg1, arg2;
477:   zend_error_handling error_handling;
478:
479:   zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
480:
481:   switch (source->type) {
482:     case SPL_FS_INFO:
483:     case SPL_FS_FILE:
484:       break;
485:     case SPL_FS_DIR:
486:       if (!source->u.dir.entry.d_name[0]) {
487:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Could not open file");
488:         zend_restore_error_handling(&error_handling);
489:         return NULL;
490:       }
491:   }
492:
493:   switch (type) {
494:     case SPL_FS_INFO:
495:       ce = ce ? ce : source->info_class;
496:
497:       if (UNEXPECTED(zend_update_class_constants(ce) != SUCCESS)) {
498:         break;
499:       }
500:
501:       intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
502:       ZVAL_OBJ(return_value, &intern->std);
503:
504:       spl_filesystem_object_get_file_name(source);
505:       if (ce->constructor->common.scope != spl_ce_SplFileInfo) {
506:         ZVAL_STRINGL(&arg1, source->file_name, source->file_name_len);
507:         zend_call_method_with_1_params(return_value, ce, &ce->constructor, "__construct", NULL, &arg1);
508:         zval_ptr_dtor(&arg1);
509:       } else {
510:         intern->file_name = estrndup(source->file_name, source->file_name_len);
511:         intern->file_name_len = source->file_name_len;
512:         intern->_path = spl_filesystem_object_get_path(source, &intern->_path_len);
513:         intern->_path = estrndup(intern->_path, intern->_path_len);
514:       }
515:       break;
516:     case SPL_FS_FILE:
517:       ce = ce ? ce : source->file_class;
518:
519:       if (UNEXPECTED(zend_update_class_constants(ce) != SUCCESS)) {
520:         break;
521:       }
522:
523:       intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
524:
525:       ZVAL_OBJ(return_value, &intern->std);
526:
527:       spl_filesystem_object_get_file_name(source);
528:
529:       if (ce->constructor->common.scope != spl_ce_SplFileObject) {
530:         ZVAL_STRINGL(&arg1, source->file_name, source->file_name_len);
531:         ZVAL_STRINGL(&arg2, "r", 1);
532:         zend_call_method_with_2_params(return_value, ce, &ce->constructor, "__construct", NULL, &arg1, &arg2);
533:         zval_ptr_dtor(&arg1);
534:         zval_ptr_dtor(&arg2);
535:       } else {
536:         intern->file_name = source->file_name;
537:         intern->file_name_len = source->file_name_len;
538:         intern->_path = spl_filesystem_object_get_path(source, &intern->_path_len);
539:         intern->_path = estrndup(intern->_path, intern->_path_len);
540:
541:         intern->u.file.open_mode = "r";
542:         intern->u.file.open_mode_len = 1;
543:
544:         if (ht && zend_parse_parameters(ht, "|sbr",
545:               &intern->u.file.open_mode, &intern->u.file.open_mode_len,
546:               &use_include_path, &intern->u.file.zcontext) == FAILURE) {
547:           zend_restore_error_handling(&error_handling);
548:           intern->u.file.open_mode = NULL;
549:           intern->file_name = NULL;
550:           zval_ptr_dtor(return_value);
551:           ZVAL_NULL(return_value);
552:           return NULL;
553:         }
554:
555:         if (spl_filesystem_file_open(intern, use_include_path, 0) == FAILURE) {
556:           zend_restore_error_handling(&error_handling);
557:           zval_ptr_dtor(return_value);
558:           ZVAL_NULL(return_value);
559:           return NULL;
560:         }
561:       }
562:       break;
563:     case SPL_FS_DIR:
564:       zend_restore_error_handling(&error_handling);
565:       zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Operation not supported");
566:       return NULL;
567:   }
568:   zend_restore_error_handling(&error_handling);
569:   return NULL;
570: } /* }}} */
571:
572: static int spl_filesystem_is_invalid_or_dot(const char * d_name) /* {{{ */
573: {
574:   return d_name[0] == '\0' || spl_filesystem_is_dot(d_name);
575: }
576: /* }}} */
577:
578: static char *spl_filesystem_object_get_pathname(spl_filesystem_object *intern, size_t *len) { /* {{{ */
579:   switch (intern->type) {
580:     case SPL_FS_INFO:
581:     case SPL_FS_FILE:
582:       *len = intern->file_name_len;
583:       return intern->file_name;
584:     case SPL_FS_DIR:
585:       if (intern->u.dir.entry.d_name[0]) {
586:         spl_filesystem_object_get_file_name(intern);
587:         *len = intern->file_name_len;
588:         return intern->file_name;
589:       }
590:   }
591:   *len = 0;
592:   return NULL;
593: }
594: /* }}} */
595:
596: static HashTable *spl_filesystem_object_get_debug_info(zval *object, int *is_temp) /* {{{ */
597: {
598:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(object);
599:   zval tmp;
600:   HashTable *rv;
601:   zend_string *pnstr;
602:   char *path;
603:   size_t  path_len;
604:   char stmp[2];
605:
606:   *is_temp = 1;
607:
608:   if (!intern->std.properties) {
609:     rebuild_object_properties(&intern->std);
610:   }
611:
612:   rv = zend_array_dup(intern->std.properties);
613:
614:   pnstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "pathName", sizeof("pathName")-1);
615:   path = spl_filesystem_object_get_pathname(intern, &path_len);
616:   ZVAL_STRINGL(&tmp, path, path_len);
617:   zend_symtable_update(rv, pnstr, &tmp);
618:   zend_string_release(pnstr);
619:
620:   if (intern->file_name) {
621:     pnstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "fileName", sizeof("fileName")-1);
622:     spl_filesystem_object_get_path(intern, &path_len);
623:
624:     if (path_len && path_len < intern->file_name_len) {
625:       ZVAL_STRINGL(&tmp, intern->file_name + path_len + 1, intern->file_name_len - (path_len + 1));
626:     } else {
627:       ZVAL_STRINGL(&tmp, intern->file_name, intern->file_name_len);
628:     }
629:     zend_symtable_update(rv, pnstr, &tmp);
630:     zend_string_release(pnstr);
631:   }
632:   if (intern->type == SPL_FS_DIR) {
633: #ifdef HAVE_GLOB
634:     pnstr = spl_gen_private_prop_name(spl_ce_DirectoryIterator, "glob", sizeof("glob")-1);
635:     if (php_stream_is(intern->u.dir.dirp ,&php_glob_stream_ops)) {
636:       ZVAL_STRINGL(&tmp, intern->_path, intern->_path_len);
637:     } else {
638:       ZVAL_FALSE(&tmp);
639:     }
640:     zend_symtable_update(rv, pnstr, &tmp);
641:     zend_string_release(pnstr);
642: #endif
643:     pnstr = spl_gen_private_prop_name(spl_ce_RecursiveDirectoryIterator, "subPathName", sizeof("subPathName")-1);
644:     if (intern->u.dir.sub_path) {
645:       ZVAL_STRINGL(&tmp, intern->u.dir.sub_path, intern->u.dir.sub_path_len);
646:     } else {
647:       ZVAL_EMPTY_STRING(&tmp);
648:     }
649:     zend_symtable_update(rv, pnstr, &tmp);
650:     zend_string_release(pnstr);
651:   }
652:   if (intern->type == SPL_FS_FILE) {
653:     pnstr = spl_gen_private_prop_name(spl_ce_SplFileObject, "openMode", sizeof("openMode")-1);
654:     ZVAL_STRINGL(&tmp, intern->u.file.open_mode, intern->u.file.open_mode_len);
655:     zend_symtable_update(rv, pnstr, &tmp);
656:     zend_string_release(pnstr);
657:     stmp[1] = '\0';
658:     stmp[0] = intern->u.file.delimiter;
659:     pnstr = spl_gen_private_prop_name(spl_ce_SplFileObject, "delimiter", sizeof("delimiter")-1);
660:     ZVAL_STRINGL(&tmp, stmp, 1);
661:     zend_symtable_update(rv, pnstr, &tmp);
662:     zend_string_release(pnstr);
663:     stmp[0] = intern->u.file.enclosure;
664:     pnstr = spl_gen_private_prop_name(spl_ce_SplFileObject, "enclosure", sizeof("enclosure")-1);
665:     ZVAL_STRINGL(&tmp, stmp, 1);
666:     zend_symtable_update(rv, pnstr, &tmp);
667:     zend_string_release(pnstr);
668:   }
669:
670:   return rv;
671: }
672: /* }}} */
673:
674: zend_function *spl_filesystem_object_get_method_check(zend_object **object, zend_string *method, const zval *key) /* {{{ */
675: {
676:   spl_filesystem_object *fsobj = spl_filesystem_from_obj(*object);
677:
678:   if (fsobj->u.dir.dirp == NULL && fsobj->orig_path == NULL) {
679:     zend_function *func;
680:     zend_string *tmp = zend_string_init("_bad_state_ex", sizeof("_bad_state_ex") - 1, 0);
681:     func = zend_get_std_object_handlers()->get_method(object, tmp, NULL);
682:     zend_string_release(tmp);
683:     return func;
684:   }
685:
686:   return zend_get_std_object_handlers()->get_method(object, method, key);
687: }
688: /* }}} */
689:
690: #define DIT_CTOR_FLAGS  0x00000001
691: #define DIT_CTOR_GLOB   0x00000002
692:
693: void spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAMETERS, zend_long ctor_flags) /* {{{ */
694: {
695:   spl_filesystem_object *intern;
696:   char *path;
697:   int parsed;
698:   size_t len;
699:   zend_long flags;
700:   zend_error_handling error_handling;
701:
702:   zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);
703:
704:   if (SPL_HAS_FLAG(ctor_flags, DIT_CTOR_FLAGS)) {
705:     flags = SPL_FILE_DIR_KEY_AS_PATHNAME|SPL_FILE_DIR_CURRENT_AS_FILEINFO;
706:     parsed = zend_parse_parameters(ZEND_NUM_ARGS(), "s|l", &path, &len, &flags);
707:   } else {
708:     flags = SPL_FILE_DIR_KEY_AS_PATHNAME|SPL_FILE_DIR_CURRENT_AS_SELF;
709:     parsed = zend_parse_parameters(ZEND_NUM_ARGS(), "s", &path, &len);
710:   }
711:   if (SPL_HAS_FLAG(ctor_flags, SPL_FILE_DIR_SKIPDOTS)) {
712:     flags |= SPL_FILE_DIR_SKIPDOTS;
713:   }
714:   if (SPL_HAS_FLAG(ctor_flags, SPL_FILE_DIR_UNIXPATHS)) {
715:     flags |= SPL_FILE_DIR_UNIXPATHS;
716:   }
717:   if (parsed == FAILURE) {
718:     zend_restore_error_handling(&error_handling);
719:     return;
720:   }
721:   if (!len) {
722:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Directory name must not be empty.");
723:     zend_restore_error_handling(&error_handling);
724:     return;
725:   }
726:
727:   intern = Z_SPLFILESYSTEM_P(getThis());
728:   if (intern->_path) {
729:     /* object is alreay initialized */
730:     zend_restore_error_handling(&error_handling);
731:     php_error_docref(NULL, E_WARNING, "Directory object is already initialized");
732:     return;
733:   }
734:   intern->flags = flags;
735: #ifdef HAVE_GLOB
736:   if (SPL_HAS_FLAG(ctor_flags, DIT_CTOR_GLOB) && strstr(path, "glob://") != path) {
737:     spprintf(&path, 0, "glob://%s", path);
738:     spl_filesystem_dir_open(intern, path);
739:     efree(path);
740:   } else
741: #endif
742:   {
743:     spl_filesystem_dir_open(intern, path);
744:
745:   }
746:
747:   intern->u.dir.is_recursive = instanceof_function(intern->std.ce, spl_ce_RecursiveDirectoryIterator) ? 1 : 0;
748:
749:   zend_restore_error_handling(&error_handling);
```

```
750: }
751: /* }}} */
752:
753: /* {{{ proto void DirectoryIterator::__construct(string path)
754:    Constructs a new dir iterator from a path. */
755: SPL_METHOD(DirectoryIterator, __construct)
756: {
757:   spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, 0);
758: }
759: /* }}} */
760:
761: /* {{{ proto void DirectoryIterator::rewind()
762:    Rewind dir back to the start */
763: SPL_METHOD(DirectoryIterator, rewind)
764: {
765:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
766:
767:   if (zend_parse_parameters_none() == FAILURE) {
768:     return;
769:   }
770:
771:   intern->u.dir.index = 0;
772:   if (intern->u.dir.dirp) {
773:     php_stream_rewinddir(intern->u.dir.dirp);
774:   }
775:   spl_filesystem_dir_read(intern);
776: }
777: /* }}} */
778:
779: /* {{{ proto string DirectoryIterator::key()
780:    Return current dir entry */
781: SPL_METHOD(DirectoryIterator, key)
782: {
783:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
784:
785:   if (zend_parse_parameters_none() == FAILURE) {
786:     return;
787:   }
788:
789:   if (intern->u.dir.dirp) {
790:     RETURN_LONG(intern->u.dir.index);
791:   } else {
792:     RETURN_FALSE;
793:   }
794: }
795: /* }}} */
796:
797: /* {{{ proto DirectoryIterator DirectoryIterator::current()
798:    Return this (needed for Iterator interface) */
799: SPL_METHOD(DirectoryIterator, current)
800: {
801:   if (zend_parse_parameters_none() == FAILURE) {
802:     return;
803:   }
804:   ZVAL_OBJ(return_value, Z_OBJ_P(getThis()));
805:   Z_ADDREF_P(return_value);
806: }
807: /* }}} */
808:
809: /* {{{ proto void DirectoryIterator::next()
810:    Move to next entry */
811: SPL_METHOD(DirectoryIterator, next)
812: {
813:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
814:   int skip_dots = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_SKIPDOTS);
815:
816:   if (zend_parse_parameters_none() == FAILURE) {
817:     return;
818:   }
819:
820:   intern->u.dir.index++;
821:   do {
822:     spl_filesystem_dir_read(intern);
823:   } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry.d_name));
824:   if (intern->file_name) {
825:     efree(intern->file_name);
826:     intern->file_name = NULL;
827:   }
828: }
829: /* }}} */
830:
831: /* {{{ proto void DirectoryIterator::seek(int position)
832:    Seek to the given position */
833: SPL_METHOD(DirectoryIterator, seek)
834: {
835:   spl_filesystem_object *intern    = Z_SPLFILESYSTEM_P(getThis());
836:   zval retval;
837:   zend_long pos;
838:
839:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &pos) == FAILURE) {
840:     return;
841:   }
842:
843:   if (intern->u.dir.index > pos) {
844:     /* we first rewind */
845:     zend_call_method_with_0_params(&EX(This), Z_OBJCE(EX(This)), &intern->u.dir.func_rewind, "rewind", NULL);
846:   }
847:
848:   while (intern->u.dir.index < pos) {
849:     int valid = 0;
850:     zend_call_method_with_0_params(&EX(This), Z_OBJCE(EX(This)), &intern->u.dir.func_valid, "valid", &retval);
851:     if (!Z_ISUNDEF(retval)) {
852:       valid = zend_is_true(&retval);
853:       zval_ptr_dtor(&retval);
854:     }
855:     if (!valid) {
856:       zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Seek position " ZEND_LONG_FMT " is out of range", pos);
857:       return;
858:     }
859:     zend_call_method_with_0_params(&EX(This), Z_OBJCE(EX(This)), &intern->u.dir.func_next, "next", NULL);
860:   }
861: } /* }}} */
862:
863: /* {{{ proto string DirectoryIterator::valid()
864:    Check whether dir contains more entries */
865: SPL_METHOD(DirectoryIterator, valid)
866: {
867:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
868:
869:   if (zend_parse_parameters_none() == FAILURE) {
870:     return;
871:   }
872:
873:   RETURN_BOOL(intern->u.dir.entry.d_name[0] != '\0');
874: }
875: /* }}} */
876:
877: /* {{{ proto string SplFileInfo::getPath()
878:    Return the path */
879: SPL_METHOD(SplFileInfo, getPath)
880: {
881:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
882:   char *path;
883:   size_t path_len;
884:
885:   if (zend_parse_parameters_none() == FAILURE) {
886:     return;
887:   }
888:
889:   path = spl_filesystem_object_get_path(intern, &path_len);
890:   RETURN_STRINGL(path, path_len);
891: }
892: /* }}} */
893:
894: /* {{{ proto string SplFileInfo::getFilename()
895:    Return filename only */
896: SPL_METHOD(SplFileInfo, getFilename)
897: {
898:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
899:   size_t path_len;
900:
901:   if (zend_parse_parameters_none() == FAILURE) {
902:     return;
903:   }
904:
905:   spl_filesystem_object_get_path(intern, &path_len);
906:
907:   if (path_len && path_len < intern->file_name_len) {
908:     RETURN_STRINGL(intern->file_name + path_len + 1, intern->file_name_len - (path_len + 1));
909:   } else {
910:     RETURN_STRINGL(intern->file_name, intern->file_name_len);
911:   }
912: }
913: /* }}} */
914:
915: /* {{{ proto string DirectoryIterator::getFilename()
916:    Return filename of current dir entry */
917: SPL_METHOD(DirectoryIterator, getFilename)
918: {
919:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
920:
921:   if (zend_parse_parameters_none() == FAILURE) {
922:     return;
923:   }
924:
925:   RETURN_STRING(intern->u.dir.entry.d_name);
926: }
927: /* }}} */
928:
929: /* {{{ proto string SplFileInfo::getExtension()
930:    Returns file extension component of path */
931: SPL_METHOD(SplFileInfo, getExtension)
932: {
933:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
934:   char *fname = NULL;
935:   const char *p;
936:   size_t flen;
937:   size_t path_len;
938:   size_t idx;
939:   zend_string *ret;
940:
941:   if (zend_parse_parameters_none() == FAILURE) {
942:     return;
943:   }
944:
945:   spl_filesystem_object_get_path(intern, &path_len);
946:
947:   if (path_len && path_len < intern->file_name_len) {
948:     fname = intern->file_name + path_len + 1;
949:     flen = intern->file_name_len - (path_len + 1);
950:   } else {
951:     fname = intern->file_name;
952:     flen = intern->file_name_len;
953:   }
954:
955:   ret = php_basename(fname, flen, NULL, 0);
956:
957:   p = zend_memrchr(ZSTR_VAL(ret), '.', ZSTR_LEN(ret));
958:   if (p) {
959:     idx = p - ZSTR_VAL(ret);
960:     RETVAL_STRINGL(ZSTR_VAL(ret) + idx + 1, ZSTR_LEN(ret) - idx - 1);
961:     zend_string_release(ret);
962:     return;
963:   } else {
964:     zend_string_release(ret);
965:     RETURN_EMPTY_STRING();
966:   }
967: }
968: /* }}}*/
969:
970: /* {{{ proto string DirectoryIterator::getExtension()
971:    Returns the file extension component of path */
972: SPL_METHOD(DirectoryIterator, getExtension)
973: {
974:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
975:   const char *p;
976:   size_t idx;
977:   zend_string *fname;
978:
979:   if (zend_parse_parameters_none() == FAILURE) {
980:     return;
981:   }
982:
983:   fname = php_basename(intern->u.dir.entry.d_name, strlen(intern->u.dir.entry.d_name), NULL, 0);
984:
985:   p = zend_memrchr(ZSTR_VAL(fname), '.', ZSTR_LEN(fname));
986:   if (p) {
987:     idx = p - ZSTR_VAL(fname);
988:     RETVAL_STRINGL(ZSTR_VAL(fname) + idx + 1, ZSTR_LEN(fname) - idx - 1);
989:     zend_string_release(fname);
990:   } else {
991:     zend_string_release(fname);
992:     RETURN_EMPTY_STRING();
993:   }
994: }
995: /* }}} */
996:
997: /* {{{ proto string SplFileInfo::getBasename([string $suffix])
998:    Returns filename component of path */
999: SPL_METHOD(SplFileInfo, getBasename)
1000: {
1001:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
1002:   char *fname, *suffix = 0;
1003:   size_t flen;
1004:   size_t slen = 0, path_len;
1005:
1006:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "|s", &suffix, &slen) == FAILURE) {
1007:     return;
1008:   }
1009:
1010:   spl_filesystem_object_get_path(intern, &path_len);
1011:
1012:   if (path_len && path_len < intern->file_name_len) {
1013:     fname = intern->file_name + path_len + 1;
1014:     flen = intern->file_name_len - (path_len + 1);
1015:   } else {
1016:     fname = intern->file_name;
1017:     flen = intern->file_name_len;
1018:   }
1019:
1020:   RETURN_STR(php_basename(fname, flen, suffix, slen));
1021: }
1022: /* }}}*/
1023:
1024: /* {{{ proto string DirectoryIterator::getBasename([string $suffix])
1025:    Returns filename component of current dir entry */
1026: SPL_METHOD(DirectoryIterator, getBasename)
1027: {
1028:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
1029:   char *suffix = 0;
1030:   size_t slen = 0;
1031:   zend_string *fname;
1032:
1033:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "|s", &suffix, &slen) == FAILURE) {
1034:     return;
1035:   }
1036:
1037:   fname = php_basename(intern->u.dir.entry.d_name, strlen(intern->u.dir.entry.d_name), suffix, slen);
1038:
1039:   RETVAL_STR(fname);
1040: }
1041: /* }}} */
1042:
1043: /* {{{ proto string SplFileInfo::getPathname()
1044:    Return path and filename */
1045: SPL_METHOD(SplFileInfo, getPathname)
1046: {
1047:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
1048:   char *path;
1049:   size_t path_len;
1050:
1051:   if (zend_parse_parameters_none() == FAILURE) {
1052:     return;
1053:   }
1054:   path = spl_filesystem_object_get_pathname(intern, &path_len);
1055:   if (path != NULL) {
1056:     RETURN_STRINGL(path, path_len);
1057:   } else {
1058:     RETURN_FALSE;
1059:   }
1060: }
1061: /* }}} */
1062:
1063: /* {{{ proto string FilesystemIterator::key()
1064:    Return getPathname() or getFilename() depending on flags */
1065: SPL_METHOD(FilesystemIterator, key)
1066: {
1067:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
1068:
1069:   if (zend_parse_parameters_none() == FAILURE) {
1070:     return;
1071:   }
1072:
1073:   if (SPL_FILE_DIR_KEY(intern, SPL_FILE_DIR_KEY_AS_FILENAME)) {
1074:     RETURN_STRING(intern->u.dir.entry.d_name);
1075:   } else {
1076:     spl_filesystem_object_get_file_name(intern);
1077:     RETURN_STRINGL(intern->file_name, intern->file_name_len);
1078:   }
1079: }
1080: /* }}} */
1081:
1082: /* {{{ proto string FilesystemIterator::current()
1083:    Return getFilename(), getFileInfo() or $this depending on flags */
1084: SPL_METHOD(FilesystemIterator, current)
1085: {
1086:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
1087:
1088:   if (zend_parse_parameters_none() == FAILURE) {
1089:     return;
1090:   }
1091:
1092:   if (SPL_FILE_DIR_CURRENT(intern, SPL_FILE_DIR_CURRENT_AS_PATHNAME)) {
1093:     spl_filesystem_object_get_file_name(intern);
1094:     RETURN_STRINGL(intern->file_name, intern->file_name_len);
1095:   } else if (SPL_FILE_DIR_CURRENT(intern, SPL_FILE_DIR_CURRENT_AS_FILEINFO)) {
1096:     spl_filesystem_object_get_file_name(intern);
1097:     spl_filesystem_object_create_type(0, intern, SPL_FS_INFO, NULL, return_value);
1098:   } else {
1099:     ZVAL_OBJ(return_value, Z_OBJ_P(getThis()));
1100:     Z_ADDREF_P(return_value);
1101:     /*RETURN_STRING(intern->u.dir.entry.d_name, 1);*/
1102:   }
1103: }
1104: /* }}} */
1105:
1106: /* {{{ proto bool DirectoryIterator::isDot()
1107:    Returns true if current entry is '.' or '..' */
1108: SPL_METHOD(DirectoryIterator, isDot)
1109: {
1110:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
1111:
1112:   if (zend_parse_parameters_none() == FAILURE) {
1113:     return;
1114:   }
1115:
1116:   RETURN_BOOL(spl_filesystem_is_dot(intern->u.dir.entry.d_name));
1117: }
1118: /* }}} */
1119:
1120: /* {{{ proto void SplFileInfo::__construct(string file_name)
1121:    Constructs a new SplFileInfo from a path. */
1122: /* When the constructor gets called the object is already created
1123:    by the engine, so we must only call 'additional' initializations.
1124:  */
1125: SPL_METHOD(SplFileInfo, __construct)
```

```
{
  spl_filesystem_object *intern;
  char *path;
  size_t len;

  if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "s", &path, &len) == FAILURE) {
    return;
  }

  intern = Z_SPLFILESYSTEM_P(getThis());

  spl_filesystem_info_set_filename(intern, path, len, 1);

  /* intern->type = SPL_FS_INFO; already set */
}
/* }}} */

/* {{{ FileInfoFunction */
#define FileInfoFunction(func_name, func_num) \
SPL_METHOD(SplFileInfo, func_name) \
{ \
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis()); \
  zend_error_handling error_handling; \
  if (zend_parse_parameters_none() == FAILURE) { \
    return; \
  } \
  \
  zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);\
  spl_filesystem_object_get_file_name(intern); \
  php_stat(intern->file_name, intern->file_name_len, func_num, return_value); \
  zend_restore_error_handling(&error_handling); \
}
/* }}} */

/* {{{ proto int SplFileInfo::getPerms()
   Get file permissions */
FileInfoFunction(getPerms, FS_PERMS)
/* }}} */

/* {{{ proto int SplFileInfo::getInode()
   Get file inode */
FileInfoFunction(getInode, FS_INODE)
/* }}} */

/* {{{ proto int SplFileInfo::getSize()
   Get file size */
FileInfoFunction(getSize, FS_SIZE)
/* }}} */

/* {{{ proto int SplFileInfo::getOwner()
   Get file owner */
FileInfoFunction(getOwner, FS_OWNER)
/* }}} */

/* {{{ proto int SplFileInfo::getGroup()
   Get file group */
FileInfoFunction(getGroup, FS_GROUP)
/* }}} */

/* {{{ proto int SplFileInfo::getATime()
   Get last access time of file */
FileInfoFunction(getATime, FS_ATIME)
/* }}} */

/* {{{ proto int SplFileInfo::getMTime()
   Get last modification time of file */
FileInfoFunction(getMTime, FS_MTIME)
/* }}} */

/* {{{ proto int SplFileInfo::getCTime()
   Get inode modification time of file */
FileInfoFunction(getCTime, FS_CTIME)
/* }}} */

/* {{{ proto string SplFileInfo::getType()
   Get file type */
FileInfoFunction(getType, FS_TYPE)
/* }}} */

/* {{{ proto bool SplFileInfo::isWritable()
   Returns true if file can be written */
FileInfoFunction(isWritable, FS_IS_W)
/* }}} */

/* {{{ proto bool SplFileInfo::isReadable()
   Returns true if file can be read */
FileInfoFunction(isReadable, FS_IS_R)
/* }}} */

/* {{{ proto bool SplFileInfo::isExecutable()
   Returns true if file is executable */
FileInfoFunction(isExecutable, FS_IS_X)
/* }}} */

/* {{{ proto bool SplFileInfo::isFile()
   Returns true if file is a regular file */
FileInfoFunction(isFile, FS_IS_FILE)
/* }}} */

/* {{{ proto bool SplFileInfo::isDir()
   Returns true if file is directory */
FileInfoFunction(isDir, FS_IS_DIR)
/* }}} */

/* {{{ proto bool SplFileInfo::isLink()
   Returns true if file is symbolic link */
FileInfoFunction(isLink, FS_IS_LINK)
/* }}} */

/* {{{ proto string SplFileInfo::getLinkTarget()
   Return the target of a symbolic link */
SPL_METHOD(SplFileInfo, getLinkTarget)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  ssize_t ret;
  char buff[MAXPATHLEN];
  zend_error_handling error_handling;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);

#if defined(PHP_WIN32) || HAVE_SYMLINK
  if (intern->file_name == NULL) {
    php_error_docref(NULL, E_WARNING, "Empty filename");
    RETURN_FALSE;
  } else if (!IS_ABSOLUTE_PATH(intern->file_name, intern->file_name_len)) {
    char expanded_path[MAXPATHLEN];
    if (!expand_filepath_with_mode(intern->file_name, expanded_path, NULL, 0, CWD_EXPAND )) {
      php_error_docref(NULL, E_WARNING, "No such file or directory");
      RETURN_FALSE;
    }
    ret = php_sys_readlink(expanded_path, buff, MAXPATHLEN - 1);
  } else {
    ret = php_sys_readlink(intern->file_name, buff,  MAXPATHLEN-1);
  }
#else
  ret = -1; /* always fail if not implemented */
#endif

  if (ret == -1) {
    zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Unable to read link %s, error: %s", intern->file_name, strerror(errno));
    RETVAL_FALSE;
  } else {
    /* Append NULL to the end of the string */
    buff[ret] = '\0';

    RETVAL_STRINGL(buff, ret);
  }
  zend_restore_error_handling(&error_handling);
}
/* }}} */

#if HAVE_REALPATH || defined(ZTS)
/* {{{ proto string SplFileInfo::getRealPath()
   Return the resolved path */
SPL_METHOD(SplFileInfo, getRealPath)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  char buff[MAXPATHLEN];
  char *filename;
  zend_error_handling error_handling;

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);

  if (intern->type == SPL_FS_DIR && !intern->file_name && intern->u.dir.entry.d_name[0]) {
    spl_filesystem_object_get_file_name(intern);
  }

  if (intern->orig_path) {
    filename = intern->orig_path;
  } else {
    filename = intern->file_name;
  }

  if (filename && VCWD_REALPATH(filename, buff)) {
#ifdef ZTS
    if (VCWD_ACCESS(buff, F_OK)) {
      RETVAL_FALSE;
    } else
#endif
    RETVAL_STRING(buff);
  } else {
    RETVAL_FALSE;
  }

  zend_restore_error_handling(&error_handling);
}
/* }}} */
#endif

/* {{{ proto void SplFileObject SplFileInfo::openFile([string mode = 'r' [, bool use_include_path  [, resource context]]])
   Open the current file */
SPL_METHOD(SplFileInfo, openFile)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  spl_filesystem_object_create_type(ZEND_NUM_ARGS(), intern, SPL_FS_FILE, NULL, return_value);
}
/* }}} */

/* {{{ proto void SplFileInfo::setFileClass([string class_name])
   Class to use in openFile() */
SPL_METHOD(SplFileInfo, setFileClass)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_class_entry *ce = spl_ce_SplFileObject;
  zend_error_handling error_handling;

  zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
    intern->file_class = ce;
  }

  zend_restore_error_handling(&error_handling);
}
/* }}} */

/* {{{ proto void SplFileInfo::setInfoClass([string class_name])
   Class to use in getFileInfo(), getPathInfo() */
SPL_METHOD(SplFileInfo, setInfoClass)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_class_entry *ce = spl_ce_SplFileInfo;
  zend_error_handling error_handling;

  zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling );

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
    intern->info_class = ce;
  }

  zend_restore_error_handling(&error_handling);
}
/* }}} */

/* {{{ proto SplFileInfo SplFileInfo::getFileInfo([string $class_name])
   Get/copy file info */
SPL_METHOD(SplFileInfo, getFileInfo)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_class_entry *ce = intern->info_class;
  zend_error_handling error_handling;

  zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
    spl_filesystem_object_create_type(ZEND_NUM_ARGS(), intern, SPL_FS_INFO, ce, return_value);
  }

  zend_restore_error_handling(&error_handling);
}
/* }}} */

/* {{{ proto SplFileInfo SplFileInfo::getPathInfo([string $class_name])
   Get/copy file info */
SPL_METHOD(SplFileInfo, getPathInfo)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_class_entry *ce = intern->info_class;
  zend_error_handling error_handling;

  zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
    size_t path_len;
    char *path = spl_filesystem_object_get_pathname(intern, &path_len);
    if (path) {
      char *dpath = estrndup(path, path_len);
      path_len = php_dirname(dpath, path_len);
      spl_filesystem_object_create_info(intern, dpath, path_len, 1, ce, return_value);
      efree(dpath);
    }
  }

  zend_restore_error_handling(&error_handling);
}
/* }}} */

/* {{{  proto void SplFileInfo::_bad_state_ex(void) */
SPL_METHOD(SplFileInfo, _bad_state_ex)
{
  zend_throw_exception_ex(spl_ce_LogicException, 0,
    "The parent constructor was not called: the object is in an "
    "invalid state ");
}
/* }}} */

/* {{{ proto void FilesystemIterator::__construct(string path [, int flags])
 Constructs a new dir iterator from a path. */
SPL_METHOD(FilesystemIterator, __construct)
{
  spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, DIT_CTOR_FLAGS | SPL_FILE_DIR_SKIPDOTS);
}
/* }}} */

/* {{{ proto void FilesystemIterator::rewind()
   Rewind dir back to the start */
SPL_METHOD(FilesystemIterator, rewind)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  int skip_dots = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_SKIPDOTS);

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  intern->u.dir.index = 0;
  if (intern->u.dir.dirp) {
    php_stream_rewinddir(intern->u.dir.dirp);
  }
  do {
    spl_filesystem_dir_read(intern);
  } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry.d_name));
}
/* }}} */

/* {{{ proto int FilesystemIterator::getFlags()
   Get handling flags */
SPL_METHOD(FilesystemIterator, getFlags)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters_none() == FAILURE) {
    return;
  }

  RETURN_LONG(intern->flags & (SPL_FILE_DIR_KEY_MODE_MASK | SPL_FILE_DIR_CURRENT_MODE_MASK | SPL_FILE_DIR_OTHERS_MASK));
} /* }}} */

/* {{{ proto void FilesystemIterator::setFlags(long $flags)
   Set handling flags */
SPL_METHOD(FilesystemIterator, setFlags)
{
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
  zend_long flags;

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &flags) == FAILURE) {
    return;
  }

  intern->flags &= ~(SPL_FILE_DIR_KEY_MODE_MASK|SPL_FILE_DIR_CURRENT_MODE_MASK|SPL_FILE_DIR_OTHERS_MASK);
  intern->flags |= ((SPL_FILE_DIR_KEY_MODE_MASK|SPL_FILE_DIR_CURRENT_MODE_MASK|SPL_FILE_DIR_OTHERS_MASK) & flags);
} /* }}} */

/* {{{ proto bool RecursiveDirectoryIterator::hasChildren([bool $allow_links = false])
   Returns whether current entry is a directory and not '.' or '..' */
SPL_METHOD(RecursiveDirectoryIterator, hasChildren)
{
  zend_bool allow_links = 0;
  spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

  if (zend_parse_parameters(ZEND_NUM_ARGS(), "|b", &allow_links) == FAILURE) {
    return;
  }
  if (spl_filesystem_is_invalid_or_dot(intern->u.dir.entry.d_name)) {
    RETURN_FALSE;
  } else {
    spl_filesystem_object_get_file_name(intern);
    if (!allow_links && !(intern->flags & SPL_FILE_DIR_FOLLOW_SYMLINKS)) {
      php_stat(intern->file_name, intern->file_name_len, FS_IS_LINK, return_value);
      if (zend_is_true(return_value)) {
        RETURN_FALSE;
      }
    }
    php_stat(intern->file_name, intern->file_name_len, FS_IS_DIR, return_value);
  }
```

```
}
/* }}} */

/* {{{ proto RecursiveDirectoryIterator DirectoryIterator::getChildren()
   Returns an iterator for the current entry if it is a directory */
SPL_METHOD(RecursiveDirectoryIterator, getChildren)
{
	zval zpath, zflags;
	spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
	spl_filesystem_object *subdir;
	char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIXPATHS) ? '/' : DEFAULT_SLASH;

	if (zend_parse_parameters_none() == FAILURE) {
		return;
	}

	spl_filesystem_object_get_file_name(intern);

	ZVAL_LONG(&zflags, intern->flags);
	ZVAL_STRINGL(&zpath, intern->file_name, intern->file_name_len);
	spl_instantiate_arg_ex2(Z_OBJCE_P(getThis()), return_value, &zpath, &zflags);
	zval_ptr_dtor(&zpath);
	zval_ptr_dtor(&zflags);

	subdir = Z_SPLFILESYSTEM_P(return_value);
	if (subdir) {
		if (intern->u.dir.sub_path && intern->u.dir.sub_path[0]) {
			subdir->u.dir.sub_path_len = spprintf(&subdir->u.dir.sub_path, 0, "%s%c%s", intern->u.dir.sub_path, slash, intern->u.dir.entry.d_name);
		} else {
			subdir->u.dir.sub_path_len = strlen(intern->u.dir.entry.d_name);
			subdir->u.dir.sub_path = estrndup(intern->u.dir.entry.d_name, subdir->u.dir.sub_path_len);
		}
		subdir->info_class = intern->info_class;
		subdir->file_class = intern->file_class;
		subdir->oth = intern->oth;
	}
}
/* }}} */

/* {{{ proto void RecursiveDirectoryIterator::getSubPath()
   Get sub path */
SPL_METHOD(RecursiveDirectoryIterator, getSubPath)
{
	spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

	if (zend_parse_parameters_none() == FAILURE) {
		return;
	}

	if (intern->u.dir.sub_path) {
		RETURN_STRINGL(intern->u.dir.sub_path, intern->u.dir.sub_path_len);
	} else {
		RETURN_EMPTY_STRING();
	}
}
/* }}} */

/* {{{ proto void RecursiveDirectoryIterator::getSubPathname()
   Get sub path and file name */
SPL_METHOD(RecursiveDirectoryIterator, getSubPathname)
{
	spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
	char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIXPATHS) ? '/' : DEFAULT_SLASH;

	if (zend_parse_parameters_none() == FAILURE) {
		return;
	}

	if (intern->u.dir.sub_path) {
		RETURN_NEW_STR(strpprintf(0, "%s%c%s", intern->u.dir.sub_path, slash, intern->u.dir.entry.d_name));
	} else {
		RETURN_STRING(intern->u.dir.entry.d_name);
	}
}
/* }}} */

/* {{{ proto int RecursiveDirectoryIterator::__construct(string path [, int flags])
 Constructs a new dir iterator from a path. */
SPL_METHOD(RecursiveDirectoryIterator, __construct)
{
	spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, DIT_CTOR_FLAGS);
}
/* }}} */

#ifdef HAVE_GLOB
/* {{{ proto int GlobIterator::__construct(string path [, int flags])
 Constructs a new dir iterator from a glob expression (no glob:// needed). */
SPL_METHOD(GlobIterator, __construct)
{
	spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, DIT_CTOR_FLAGS|DIT_CTOR_GLOB);
}
/* }}} */

/* {{{ proto int GlobIterator::cont()
   Return the number of directories and files found by globbing */
SPL_METHOD(GlobIterator, count)
{
	spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());

	if (zend_parse_parameters_none() == FAILURE) {
		return;
	}

	if (intern->u.dir.dirp && php_stream_is(intern->u.dir.dirp ,&php_glob_stream_ops)) {
		RETURN_LONG(php_glob_stream_get_count(intern->u.dir.dirp, NULL));
	} else {
		/* should not happen */
		php_error_docref(NULL, E_ERROR, "GlobIterator lost glob state");
	}
}
/* }}} */
#endif /* HAVE_GLOB */

/* {{{ forward declarations to the iterator handlers */
static void spl_filesystem_dir_it_dtor(zend_object_iterator *iter);
static int spl_filesystem_dir_it_valid(zend_object_iterator *iter);
static zval *spl_filesystem_dir_it_current_data(zend_object_iterator *iter);
static void spl_filesystem_dir_it_current_key(zend_object_iterator *iter, zval *key);
static void spl_filesystem_dir_it_move_forward(zend_object_iterator *iter);
static void spl_filesystem_dir_it_rewind(zend_object_iterator *iter);

/* iterator handler table */
static const zend_object_iterator_funcs spl_filesystem_dir_it_funcs = {
	spl_filesystem_dir_it_dtor,
	spl_filesystem_dir_it_valid,
	spl_filesystem_dir_it_current_data,
	spl_filesystem_dir_it_current_key,
	spl_filesystem_dir_it_move_forward,
	spl_filesystem_dir_it_rewind,
	NULL
};
/* }}} */

/* {{{ spl_ce_dir_get_iterator */
zend_object_iterator *spl_filesystem_dir_get_iterator(zend_class_entry *ce, zval *object, int by_ref)
{
	spl_filesystem_iterator *iterator;
	spl_filesystem_object *dir_object;

	if (by_ref) {
		zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
		return NULL;
	}
	dir_object = Z_SPLFILESYSTEM_P(object);
	iterator = spl_filesystem_object_to_iterator(dir_object);
	ZVAL_COPY(&iterator->intern.data, object);
	iterator->intern.funcs = &spl_filesystem_dir_it_funcs;
	/* ->current must be initialized; rewind doesn't set it and valid
	 * doesn't check whether it's set */
	iterator->current = *object;

	return &iterator->intern;
}
/* }}} */

/* {{{ spl_filesystem_dir_it_dtor */
static void spl_filesystem_dir_it_dtor(zend_object_iterator *iter)
{
	spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;

	if (!Z_ISUNDEF(iterator->intern.data)) {
		zval *object = &iterator->intern.data;
		zval_ptr_dtor(object);
	}
	/* Otherwise we were called from the owning object free storage handler as
	 * it sets iterator->intern.data to IS_UNDEF.
	 * We don't even need to destroy iterator->current as we didn't add a
	 * reference to it in move_forward or get_iterator */
}
/* }}} */

/* {{{ spl_filesystem_dir_it_valid */
static int spl_filesystem_dir_it_valid(zend_object_iterator *iter)
{
	spl_filesystem_object *object = spl_filesystem_iterator_to_object((spl_filesystem_iterator *)iter);

	return object->u.dir.entry.d_name[0] != '\0' ? SUCCESS : FAILURE;
}
/* }}} */

/* {{{ spl_filesystem_dir_it_current_data */
static zval *spl_filesystem_dir_it_current_data(zend_object_iterator *iter)
{
	spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;

	return &iterator->current;
}
/* }}} */

/* {{{ spl_filesystem_dir_it_current_key */
static void spl_filesystem_dir_it_current_key(zend_object_iterator *iter, zval *key)
{
	spl_filesystem_object *object = spl_filesystem_iterator_to_object((spl_filesystem_iterator *)iter);

	ZVAL_LONG(key, object->u.dir.index);
}
/* }}} */

/* {{{ spl_filesystem_dir_it_move_forward */
static void spl_filesystem_dir_it_move_forward(zend_object_iterator *iter)
{
	spl_filesystem_object *object = spl_filesystem_iterator_to_object((spl_filesystem_iterator *)iter);

	object->u.dir.index++;
	spl_filesystem_dir_read(object);
	if (object->file_name) {
		efree(object->file_name);
		object->file_name = NULL;
	}
}
/* }}} */

/* {{{ spl_filesystem_dir_it_rewind */
static void spl_filesystem_dir_it_rewind(zend_object_iterator *iter)
{
	spl_filesystem_object *object = spl_filesystem_iterator_to_object((spl_filesystem_iterator *)iter);

	object->u.dir.index = 0;
	if (object->u.dir.dirp) {
		php_stream_rewinddir(object->u.dir.dirp);
	}
	spl_filesystem_dir_read(object);
}
/* }}} */

/* {{{ spl_filesystem_tree_it_dtor */
static void spl_filesystem_tree_it_dtor(zend_object_iterator *iter)
{
	spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;

	if (!Z_ISUNDEF(iterator->intern.data)) {
		zval *object = &iterator->intern.data;
		zval_ptr_dtor(object);
	} else {
		if (!Z_ISUNDEF(iterator->current)) {
			zval_ptr_dtor(&iterator->current);
			ZVAL_UNDEF(&iterator->current);
		}
	}
}
/* }}} */

/* {{{ spl_filesystem_tree_it_current_data */
static zval *spl_filesystem_tree_it_current_data(zend_object_iterator *iter)
{
	spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
	spl_filesystem_object   *object   = spl_filesystem_iterator_to_object(iterator);

	if (SPL_FILE_DIR_CURRENT(object, SPL_FILE_DIR_CURRENT_AS_PATHNAME)) {
		if (Z_ISUNDEF(iterator->current)) {
			spl_filesystem_object_get_file_name(object);
			ZVAL_STRINGL(&iterator->current, object->file_name, object->file_name_len);
		}
		return &iterator->current;
	} else if (SPL_FILE_DIR_CURRENT(object, SPL_FILE_DIR_CURRENT_AS_FILEINFO)) {
		if (Z_ISUNDEF(iterator->current)) {
			spl_filesystem_object_get_file_name(object);
			spl_filesystem_object_create_type(0, object, SPL_FS_INFO, NULL, &iterator->current);
		}
		return &iterator->current;
	} else {
		return &iterator->intern.data;
	}
}
/* }}} */

/* {{{ spl_filesystem_tree_it_current_key */
static void spl_filesystem_tree_it_current_key(zend_object_iterator *iter, zval *key)
{
	spl_filesystem_object *object = spl_filesystem_iterator_to_object((spl_filesystem_iterator *)iter);

	if (SPL_FILE_DIR_KEY(object, SPL_FILE_DIR_KEY_AS_FILENAME)) {
		ZVAL_STRING(key, object->u.dir.entry.d_name);
	} else {
		spl_filesystem_object_get_file_name(object);
		ZVAL_STRINGL(key, object->file_name, object->file_name_len);
	}
}
/* }}} */

/* {{{ spl_filesystem_tree_it_move_forward */
static void spl_filesystem_tree_it_move_forward(zend_object_iterator *iter)
{
	spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
	spl_filesystem_object   *object   = spl_filesystem_iterator_to_object(iterator);

	object->u.dir.index++;
	do {
		spl_filesystem_dir_read(object);
	} while (spl_filesystem_is_dot(object->u.dir.entry.d_name));
	if (object->file_name) {
		efree(object->file_name);
		object->file_name = NULL;
	}
	if (!Z_ISUNDEF(iterator->current)) {
		zval_ptr_dtor(&iterator->current);
		ZVAL_UNDEF(&iterator->current);
	}
}
/* }}} */

/* {{{ spl_filesystem_tree_it_rewind */
static void spl_filesystem_tree_it_rewind(zend_object_iterator *iter)
{
	spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
	spl_filesystem_object   *object   = spl_filesystem_iterator_to_object(iterator);

	object->u.dir.index = 0;
	if (object->u.dir.dirp) {
		php_stream_rewinddir(object->u.dir.dirp);
	}
	do {
		spl_filesystem_dir_read(object);
	} while (spl_filesystem_is_dot(object->u.dir.entry.d_name));
	if (!Z_ISUNDEF(iterator->current)) {
		zval_ptr_dtor(&iterator->current);
		ZVAL_UNDEF(&iterator->current);
	}
}
/* }}} */

/* {{{ iterator handler table */
static const zend_object_iterator_funcs spl_filesystem_tree_it_funcs = {
	spl_filesystem_tree_it_dtor,
	spl_filesystem_dir_it_valid,
	spl_filesystem_tree_it_current_data,
	spl_filesystem_tree_it_current_key,
	spl_filesystem_tree_it_move_forward,
	spl_filesystem_tree_it_rewind,
	NULL
};
/* }}} */

/* {{{ spl_ce_dir_get_iterator */
zend_object_iterator *spl_filesystem_tree_get_iterator(zend_class_entry *ce, zval *object, int by_ref)
{
	spl_filesystem_iterator *iterator;
	spl_filesystem_object *dir_object;

	if (by_ref) {
		zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
		return NULL;
	}
	dir_object = Z_SPLFILESYSTEM_P(object);
	iterator = spl_filesystem_object_to_iterator(dir_object);

	ZVAL_COPY(&iterator->intern.data, object);
	iterator->intern.funcs = &spl_filesystem_tree_it_funcs;

	return &iterator->intern;
}
/* }}} */

/* {{{ spl_filesystem_object_cast */
static int spl_filesystem_object_cast(zval *readobj, zval *writeobj, int type)
{
	spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(readobj);

	if (type == IS_STRING) {
		if (Z_OBJCE_P(readobj)->__tostring) {
			return std_object_handlers.cast_object(readobj, writeobj, type);
		}

		switch (intern->type) {
		case SPL_FS_INFO:
		case SPL_FS_FILE:
			ZVAL_STRINGL(writeobj, intern->file_name, intern->file_name_len);
			return SUCCESS;
		case SPL_FS_DIR:
			ZVAL_STRING(writeobj, intern->u.dir.entry.d_name);
			return SUCCESS;
		}
	} else if (type == _IS_BOOL) {
		ZVAL_TRUE(writeobj);
		return SUCCESS;
	}
```

```
1878:   ZVAL_NULL(writeobj);
1879:   return FAILURE;
1880: }
1881: /* }}} */
1882:
1883: /* {{{ declare method parameters */
1884: /* supply a name and default to call by parameter */
1885: ZEND_BEGIN_ARG_INFO(arginfo_info___construct, 0)
1886:   ZEND_ARG_INFO(0, file_name)
1887: ZEND_END_ARG_INFO()
1888:
1889: ZEND_BEGIN_ARG_INFO_EX(arginfo_info_openFile, 0, 0, 0)
1890:   ZEND_ARG_INFO(0, open_mode)
1891:   ZEND_ARG_INFO(0, use_include_path)
1892:   ZEND_ARG_INFO(0, context)
1893: ZEND_END_ARG_INFO()
1894:
1895: ZEND_BEGIN_ARG_INFO_EX(arginfo_info_optinalFileClass, 0, 0, 0)
1896:   ZEND_ARG_INFO(0, class_name)
1897: ZEND_END_ARG_INFO()
1898:
1899: ZEND_BEGIN_ARG_INFO_EX(arginfo_optinalSuffix, 0, 0, 0)
1900:   ZEND_ARG_INFO(0, suffix)
1901: ZEND_END_ARG_INFO()
1902:
1903: ZEND_BEGIN_ARG_INFO(arginfo_splfileinfo_void, 0)
1904: ZEND_END_ARG_INFO()
1905:
1906: /* the method table */
1907: /* each method can have its own parameters and visibility */
1908: static const zend_function_entry spl_SplFileInfo_functions[] = {
1909:   SPL_ME(SplFileInfo,       __construct,   arginfo_info___construct, ZEND_ACC_PUBLIC)
1910:   SPL_ME(SplFileInfo,       getPath,       arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1911:   SPL_ME(SplFileInfo,       getFilename,   arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1912:   SPL_ME(SplFileInfo,       getExtension,  arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1913:   SPL_ME(SplFileInfo,       getBasename,   arginfo_optinalSuffix, ZEND_ACC_PUBLIC)
1914:   SPL_ME(SplFileInfo,       getPathname,   arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1915:   SPL_ME(SplFileInfo,       getPerms,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1916:   SPL_ME(SplFileInfo,       getInode,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1917:   SPL_ME(SplFileInfo,       getSize,       arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1918:   SPL_ME(SplFileInfo,       getOwner,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1919:   SPL_ME(SplFileInfo,       getGroup,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1920:   SPL_ME(SplFileInfo,       getATime,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1921:   SPL_ME(SplFileInfo,       getMTime,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1922:   SPL_ME(SplFileInfo,       getCTime,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1923:   SPL_ME(SplFileInfo,       getType,       arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1924:   SPL_ME(SplFileInfo,       isWritable,    arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1925:   SPL_ME(SplFileInfo,       isReadable,    arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1926:   SPL_ME(SplFileInfo,       isExecutable,  arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1927:   SPL_ME(SplFileInfo,       isFile,        arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1928:   SPL_ME(SplFileInfo,       isDir,         arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1929:   SPL_ME(SplFileInfo,       isLink,        arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1930:   SPL_ME(SplFileInfo,       getLinkTarget, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1931: #if HAVE_REALPATH || defined(ZTS)
1932:   SPL_ME(SplFileInfo,       getRealPath,   arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1933: #endif
1934:   SPL_ME(SplFileInfo,       getFileInfo,   arginfo_info_optinalFileClass, ZEND_ACC_PUBLIC)
1935:   SPL_ME(SplFileInfo,       getPathInfo,   arginfo_info_optinalFileClass, ZEND_ACC_PUBLIC)
1936:   SPL_ME(SplFileInfo,       openFile,      arginfo_info_openFile,     ZEND_ACC_PUBLIC)
1937:   SPL_ME(SplFileInfo,       setFileClass,  arginfo_info_optinalFileClass, ZEND_ACC_PUBLIC)
1938:   SPL_ME(SplFileInfo,       setInfoClass,  arginfo_info_optinalFileClass, ZEND_ACC_PUBLIC)
1939:   SPL_ME(SplFileInfo,       _bad_state_ex, NULL,                  ZEND_ACC_PUBLIC|ZEND_ACC_FINAL)
1940:   SPL_MA(SplFileInfo,       __toString, SplFileInfo, getPathname, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1941:   PHP_FE_END
1942: };
1943:
1944: ZEND_BEGIN_ARG_INFO(arginfo_dir___construct, 0)
1945:   ZEND_ARG_INFO(0, path)
1946: ZEND_END_ARG_INFO()
1947:
1948: ZEND_BEGIN_ARG_INFO(arginfo_dir_it_seek, 0)
1949:   ZEND_ARG_INFO(0, position)
1950: ZEND_END_ARG_INFO();
1951:
1952: /* the method table */
1953: /* each method can have its own parameters and visibility */
1954: static const zend_function_entry spl_DirectoryIterator_functions[] = {
1955:   SPL_ME(DirectoryIterator, __construct,   arginfo_dir___construct, ZEND_ACC_PUBLIC)
1956:   SPL_ME(DirectoryIterator, getFilename,   arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1957:   SPL_ME(DirectoryIterator, getExtension,  arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1958:   SPL_ME(DirectoryIterator, getBasename,   arginfo_optinalSuffix, ZEND_ACC_PUBLIC)
1959:   SPL_ME(DirectoryIterator, isDot,         arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1960:   SPL_ME(DirectoryIterator, rewind,        arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1961:   SPL_ME(DirectoryIterator, valid,         arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1962:   SPL_ME(DirectoryIterator, key,           arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1963:   SPL_ME(DirectoryIterator, current,       arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1964:   SPL_ME(DirectoryIterator, next,          arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1965:   SPL_ME(DirectoryIterator, seek,          arginfo_dir_it_seek, ZEND_ACC_PUBLIC)
1966:   SPL_MA(DirectoryIterator, __toString, DirectoryIterator, getFilename, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1967:   PHP_FE_END
1968: };
1969:
1970: ZEND_BEGIN_ARG_INFO_EX(arginfo_r_dir___construct, 0, 0, 1)
1971:   ZEND_ARG_INFO(0, path)
1972:   ZEND_ARG_INFO(0, flags)
1973: ZEND_END_ARG_INFO()
1974:
1975: ZEND_BEGIN_ARG_INFO_EX(arginfo_r_dir_hasChildren, 0, 0, 0)
1976:   ZEND_ARG_INFO(0, allow_links)
1977: ZEND_END_ARG_INFO()
1978:
1979: ZEND_BEGIN_ARG_INFO_EX(arginfo_r_dir_setFlags, 0, 0, 0)
1980:   ZEND_ARG_INFO(0, flags)
1981: ZEND_END_ARG_INFO()
1982:
1983: static const zend_function_entry spl_FilesystemIterator_functions[] = {
1984:   SPL_ME(FilesystemIterator, __construct,   arginfo_r_dir___construct, ZEND_ACC_PUBLIC)
1985:   SPL_ME(FilesystemIterator, rewind,        arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1986:   SPL_ME(DirectoryIterator,  next,          arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1987:   SPL_ME(FilesystemIterator, key,           arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1988:   SPL_ME(FilesystemIterator, current,       arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1989:   SPL_ME(FilesystemIterator, getFlags,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1990:   SPL_ME(FilesystemIterator, setFlags,      arginfo_r_dir_setFlags, ZEND_ACC_PUBLIC)
1991:   PHP_FE_END
1992: };
1993:
1994: static const zend_function_entry spl_RecursiveDirectoryIterator_functions[] = {
1995:   SPL_ME(RecursiveDirectoryIterator, __construct,   arginfo_r_dir___construct, ZEND_ACC_PUBLIC)
1996:   SPL_ME(RecursiveDirectoryIterator, hasChildren,   arginfo_r_dir_hasChildren, ZEND_ACC_PUBLIC)
1997:   SPL_ME(RecursiveDirectoryIterator, getChildren,   arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1998:   SPL_ME(RecursiveDirectoryIterator, getSubPath,    arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
1999:   SPL_ME(RecursiveDirectoryIterator, getSubPathname,arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
2000:   PHP_FE_END
2001: };
2002:
2003: #ifdef HAVE_GLOB
2004: static const zend_function_entry spl_GlobIterator_functions[] = {
2005:   SPL_ME(GlobIterator, __construct,   arginfo_r_dir___construct, ZEND_ACC_PUBLIC)
2006:   SPL_ME(GlobIterator, count,         arginfo_splfileinfo_void,  ZEND_ACC_PUBLIC)
2007:   PHP_FE_END
2008: };
2009: #endif
2010: /* }}} */
2011:
2012: static int spl_filesystem_file_read(spl_filesystem_object *intern, int silent) /* {{{ */
2013: {
2014:   char *buf;
2015:   size_t line_len = 0;
2016:   zend_long line_add = (intern->u.file.current_line || !Z_ISUNDEF(intern->u.file.current_zval)) ? 1 : 0;
2017:
2018:   spl_filesystem_file_free_line(intern);
2019:
2020:   if (php_stream_eof(intern->u.file.stream)) {
2021:     if (!silent) {
2022:       zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot read from file %s", intern->file_name);
2023:     }
2024:     return FAILURE;
2025:   }
2026:
2027:   if (intern->u.file.max_line_len > 0) {
2028:     buf = safe_emalloc((intern->u.file.max_line_len + 1), sizeof(char), 0);
2029:     if (php_stream_get_line(intern->u.file.stream, buf, intern->u.file.max_line_len + 1, &line_len) == NULL) {
2030:       efree(buf);
2031:       buf = NULL;
2032:     } else {
2033:       buf[line_len] = '\0';
2034:     }
2035:   } else {
2036:     buf = php_stream_get_line(intern->u.file.stream, NULL, 0, &line_len);
2037:   }
2038:
2039:   if (!buf) {
2040:     intern->u.file.current_line = estrdup("");
2041:     intern->u.file.current_line_len = 0;
2042:   } else {
2043:     if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_DROP_NEW_LINE)) {
2044:       line_len = strcspn(buf, "\r\n");
2045:       buf[line_len] = '\0';
2046:     }
2047:
2048:     intern->u.file.current_line = buf;
2049:     intern->u.file.current_line_len = line_len;
2050:   }
2051:   intern->u.file.current_line_num += line_add;
2052:
2053:   return SUCCESS;
2054: } /* }}} */
2055:
2056: static int spl_filesystem_file_call(spl_filesystem_object *intern, zend_function *func_ptr, int pass_num_args, zval *return_value, zval *arg2) /* {{{ *
/
2057: {
2058:   zend_fcall_info fci;
2059:   zend_fcall_info_cache fcic;
2060:   zval *zresource_ptr = &intern->u.file.zresource, retval;
2061:   int result;
2062:   int num_args = pass_num_args + (arg2 ? 2 : 1);
2063:
2064:   zval *params = (zval*)safe_emalloc(num_args, sizeof(zval), 0);
2065:
2066:   params[0] = *zresource_ptr;
2067:
2068:   if (arg2) {
2069:     params[1] = *arg2;
2070:   }
2071:
2072:   if (zend_get_parameters_array_ex(pass_num_args, params + (arg2 ? 2 : 1)) != SUCCESS) {
2073:     efree(params);
2074:     WRONG_PARAM_COUNT_WITH_RETVAL(FAILURE);
2075:   }
2076:
2077:   ZVAL_UNDEF(&retval);
2078:
2079:   fci.size = sizeof(fci);
2080:   fci.object = NULL;
2081:   fci.retval = &retval;
2082:   fci.param_count = num_args;
2083:   fci.params = params;
2084:   fci.no_separation = 1;
2085:   ZVAL_STR(&fci.function_name, func_ptr->common.function_name);
2086:
2087:   fcic.function_handler = func_ptr;
2088:   fcic.calling_scope = NULL;
2089:   fcic.called_scope = NULL;
2090:   fcic.object = NULL;
2091:
2092:   result = zend_call_function(&fci, &fcic);
2093:
2094:   if (result == FAILURE || Z_ISUNDEF(retval)) {
2095:     RETVAL_FALSE;
2096:   } else {
2097:     ZVAL_ZVAL(return_value, &retval, 0, 0);
2098:   }
2099:
2100:   efree(params);
2101:   return result;
2102: /* }}} */
2103:
2104: #define FileFunctionCall(func_name, pass_num_args, arg2) /* {{{ */ \
2105: { \
2106:   zend_function *func_ptr; \
2107:   func_ptr = (zend_function *)zend_hash_str_find_ptr(EG(function_table), #func_name, sizeof(#func_name) - 1); \
2108:   if (func_ptr == NULL) { \
2109:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Internal error, function '%s' not found. Please report", #func_name); \
2110:     return; \
2111:   } \
2112:   spl_filesystem_file_call(intern, func_ptr, pass_num_args, return_value, arg2); \
2113: } /* }}} */
2114:
2115: static int spl_filesystem_file_read_csv(spl_filesystem_object *intern, char delimiter, char enclosure, char escape, zval *return_value) /* {{{ */
2116: {
2117:   int ret = SUCCESS;
2118:   zval *value;
2119:
2120:   do {
2121:     ret = spl_filesystem_file_read(intern, 1);
2122:   } while (ret == SUCCESS && !intern->u.file.current_line_len && SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_SKIP_EMPTY));
2123:
2124:   if (ret == SUCCESS) {
2125:     size_t buf_len = intern->u.file.current_line_len;
2126:     char *buf = estrndup(intern->u.file.current_line, buf_len);
2127:
2128:     if (!Z_ISUNDEF(intern->u.file.current_zval)) {
2129:       zval_ptr_dtor(&intern->u.file.current_zval);
2130:       ZVAL_UNDEF(&intern->u.file.current_zval);
2131:     }
2132:
2133:     php_fgetcsv(intern->u.file.stream, delimiter, enclosure, escape, buf_len, buf, &intern->u.file.current_zval);
2134:     if (return_value) {
2135:       zval_ptr_dtor(return_value);
2136:       value = &intern->u.file.current_zval;
2137:       ZVAL_DEREF(value);
2138:       ZVAL_COPY(return_value, value);
2139:     }
2140:   }
2141:   return ret;
2142: }
2143: /* }}} */
2144:
2145: static int spl_filesystem_file_read_line_ex(zval * this_ptr, spl_filesystem_object *intern, int silent) /* {{{ */
2146: {
2147:   zval retval;
2148:
2149:   /* 1) use fgetcsv? 2) overloaded call the function, 3) do it directly */
2150:   if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV) || intern->u.file.func_getCurr->common.scope != spl_ce_SplFileObject) {
2151:     if (php_stream_eof(intern->u.file.stream)) {
2152:       if (!silent) {
2153:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot read from file %s", intern->file_name);
2154:       }
2155:       return FAILURE;
2156:     }
2157:     if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV)) {
2158:       return spl_filesystem_file_read_csv(intern, intern->u.file.delimiter, intern->u.file.enclosure, intern->u.file.escape, NULL);
2159:     } else {
2160:       zend_execute_data *execute_data = EG(current_execute_data);
2161:       zend_call_method_with_0_params(this_ptr, Z_OBJCE(EX(This)), &intern->u.file.func_getCurr, "getCurrentLine", &retval);
2162:     }
2163:     if (!Z_ISUNDEF(retval)) {
2164:       if (intern->u.file.current_line || !Z_ISUNDEF(intern->u.file.current_zval)) {
2165:         intern->u.file.current_line_num++;
2166:       }
2167:       spl_filesystem_file_free_line(intern);
2168:       if (Z_TYPE(retval) == IS_STRING) {
2169:         intern->u.file.current_line = estrndup(Z_STRVAL(retval), Z_STRLEN(retval));
2170:         intern->u.file.current_line_len = Z_STRLEN(retval);
2171:       } else {
2172:         zval *value = &retval;
2173:
2174:         ZVAL_DEREF(value);
2175:         ZVAL_COPY(&intern->u.file.current_zval, value);
2176:       }
2177:       zval_ptr_dtor(&retval);
2178:       return SUCCESS;
2179:     } else {
2180:       return FAILURE;
2181:     }
2182:   } else {
2183:     return spl_filesystem_file_read(intern, silent);
2184:   }
2185: } /* }}} */
2186:
2187: static int spl_filesystem_file_is_empty_line(spl_filesystem_object *intern) /* {{{ */
2188: {
2189:   if (intern->u.file.current_line) {
2190:     return intern->u.file.current_line_len == 0;
2191:   } else if (!Z_ISUNDEF(intern->u.file.current_zval)) {
2192:     switch(Z_TYPE(intern->u.file.current_zval)) {
2193:       case IS_STRING:
2194:         return Z_STRLEN(intern->u.file.current_zval) == 0;
2195:       case IS_ARRAY:
2196:         if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV)
2197:             && zend_hash_num_elements(Z_ARRVAL(intern->u.file.current_zval)) == 1) {
2198:           uint32_t idx = 0;
2199:           zval *first;
2200:
2201:           while (Z_ISUNDEF(Z_ARRVAL(intern->u.file.current_zval)->arData[idx].val)) {
2202:             idx++;
2203:           }
2204:           first = &Z_ARRVAL(intern->u.file.current_zval)->arData[idx].val;
2205:           return Z_TYPE_P(first) == IS_STRING && Z_STRLEN_P(first) == 0;
2206:         }
2207:         return zend_hash_num_elements(Z_ARRVAL(intern->u.file.current_zval)) == 0;
2208:       case IS_NULL:
2209:         return 1;
2210:       default:
2211:         return 0;
2212:     }
2213:   } else {
2214:     return 1;
2215:   }
2216: }
2217: /* }}} */
2218:
2219: static int spl_filesystem_file_read_line(zval * this_ptr, spl_filesystem_object *intern, int silent) /* {{{ */
2220: {
2221:   int ret = spl_filesystem_file_read_line_ex(this_ptr, intern, silent);
2222:
2223:   while (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_SKIP_EMPTY) && ret == SUCCESS && spl_filesystem_file_is_empty_line(intern)) {
2224:     spl_filesystem_file_free_line(intern);
2225:     ret = spl_filesystem_file_read_line_ex(this_ptr, intern, silent);
2226:   }
2227:
2228:   return ret;
2229: }
2230: /* }}} */
2231:
2232: static void spl_filesystem_file_rewind(zval * this_ptr, spl_filesystem_object *intern) /* {{{ */
2233: {
2234:   if(!intern->u.file.stream) {
2235:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2236:     return;
2237:   }
2238:   if (-1 == php_stream_rewind(intern->u.file.stream)) {
2239:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot rewind file %s", intern->file_name);
2240:   } else {
2241:     spl_filesystem_file_free_line(intern);
2242:     intern->u.file.current_line_num = 0;
2243:   }
2244:   if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_AHEAD)) {
2245:     spl_filesystem_file_read_line(this_ptr, intern, 1);
2246:   }
2247: } /* }}} */
2248:
2249: /* {{{ proto void SplFileObject::__construct(string filename [, string mode = 'r' [, bool use_include_path  [, resource context]]])
2250:    Construct a new file object */
2251: SPL_METHOD(SplFileObject, __construct)
2252: {
```

```
2253:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2254:   zend_bool use_include_path = 0;
2255:   char *p1, *p2;
2256:   char *tmp_path;
2257:   size_t   tmp_path_len;
2258:   zend_error_handling error_handling;
2259:
2260:   intern->u.file.open_mode = NULL;
2261:   intern->u.file.open_mode_len = 0;
2262:
2263:   if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "p|sbr!",
2264:       &intern->file_name, &intern->file_name_len,
2265:       &intern->u.file.open_mode, &intern->u.file.open_mode_len,
2266:       &use_include_path, &intern->u.file.zcontext) == FAILURE) {
2267:     intern->u.file.open_mode = NULL;
2268:     intern->file_name = NULL;
2269:     return;
2270:   }
2271:
2272:   if (intern->u.file.open_mode == NULL) {
2273:     intern->u.file.open_mode = "r";
2274:     intern->u.file.open_mode_len = 1;
2275:   }
2276:
2277:   zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
2278:
2279:   if (spl_filesystem_file_open(intern, use_include_path, 0) == SUCCESS) {
2280:     tmp_path_len = strlen(intern->u.file.stream->orig_path);
2281:
2282:     if (tmp_path_len > 1 && IS_SLASH_AT(intern->u.file.stream->orig_path, tmp_path_len-1)) {
2283:       tmp_path_len--;
2284:     }
2285:
2286:     tmp_path = estrndup(intern->u.file.stream->orig_path, tmp_path_len);
2287:
2288:     p1 = strrchr(tmp_path, '/');
2289: #if defined(PHP_WIN32)
2290:     p2 = strrchr(tmp_path, '\\');
2291: #else
2292:     p2 = 0;
2293: #endif
2294:     if (p1 || p2) {
2295:       intern->_path_len = ((p1 > p2 ? p1 : p2) - tmp_path);
2296:     } else {
2297:       intern->_path_len = 0;
2298:     }
2299:
2300:     efree(tmp_path);
2301:
2302:     intern->_path = estrndup(intern->u.file.stream->orig_path, intern->_path_len);
2303:   }
2304:
2305:   zend_restore_error_handling(&error_handling);
2306:
2307: } /* }}} */
2308:
2309: /* {{{ proto void SplTempFileObject::__construct([int max_memory])
2310:    Construct a new temp file object */
2311: SPL_METHOD(SplTempFileObject, __construct)
2312: {
2313:   zend_long max_memory = PHP_STREAM_MAX_MEM;
2314:   char tmp_fname[48];
2315:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2316:   zend_error_handling error_handling;
2317:
2318:   if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "|l", &max_memory) == FAILURE) {
2319:     return;
2320:   }
2321:
2322:   if (max_memory < 0) {
2323:     intern->file_name = "php://memory";
2324:     intern->file_name_len = 12;
2325:   } else if (ZEND_NUM_ARGS()) {
2326:     intern->file_name_len = slprintf(tmp_fname, sizeof(tmp_fname), "php://temp/maxmemory:" ZEND_LONG_FMT, max_memory);
2327:     intern->file_name = tmp_fname;
2328:   } else {
2329:     intern->file_name = "php://temp";
2330:     intern->file_name_len = 10;
2331:   }
2332:   intern->u.file.open_mode = "wb";
2333:   intern->u.file.open_mode_len = 1;
2334:
2335:   zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
2336:   if (spl_filesystem_file_open(intern, 0, 0) == SUCCESS) {
2337:     intern->_path_len = 0;
2338:     intern->_path = estrndup("", 0);
2339:   }
2340:   zend_restore_error_handling(&error_handling);
2341: } /* }}} */
2342:
2343: /* {{{ proto void SplFileObject::rewind()
2344:    Rewind the file and read the first line */
2345: SPL_METHOD(SplFileObject, rewind)
2346: {
2347:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2348:
2349:   if (zend_parse_parameters_none() == FAILURE) {
2350:     return;
2351:   }
2352:
2353:   spl_filesystem_file_rewind(getThis(), intern);
2354: } /* }}} */
2355:
2356: /* {{{ proto void SplFileObject::eof()
2357:    Return whether end of file is reached */
2358: SPL_METHOD(SplFileObject, eof)
2359: {
2360:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2361:
2362:   if (zend_parse_parameters_none() == FAILURE) {
2363:     return;
2364:   }
2365:
2366:   if(!intern->u.file.stream) {
2367:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2368:     return;
2369:   }
2370:
2371:   RETURN_BOOL(php_stream_eof(intern->u.file.stream));
2372: } /* }}} */
2373:
2374: /* {{{ proto void SplFileObject::valid()
2375:    Return !eof() */
2376: SPL_METHOD(SplFileObject, valid)
2377: {
2378:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2379:
2380:   if (zend_parse_parameters_none() == FAILURE) {
2381:     return;
2382:   }
2383:
2384:   if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_AHEAD)) {
2385:     RETURN_BOOL(intern->u.file.current_line || !Z_ISUNDEF(intern->u.file.current_zval));
2386:   } else {
2387:     if(!intern->u.file.stream) {
2388:       RETURN_FALSE;
2389:     }
2390:     RETVAL_BOOL(!php_stream_eof(intern->u.file.stream));
2391:   }
2392: } /* }}} */
2393:
2394: /* {{{ proto string SplFileObject::fgets()
2395:    Return next line from file */
2396: SPL_METHOD(SplFileObject, fgets)
2397: {
2398:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2399:
2400:   if (zend_parse_parameters_none() == FAILURE) {
2401:     return;
2402:   }
2403:
2404:   if(!intern->u.file.stream) {
2405:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2406:     return;
2407:   }
2408:
2409:   if (spl_filesystem_file_read(intern, 0) == FAILURE) {
2410:     RETURN_FALSE;
2411:   }
2412:   RETURN_STRINGL(intern->u.file.current_line, intern->u.file.current_line_len);
2413: } /* }}} */
2414:
2415: /* {{{ proto string SplFileObject::current()
2416:    Return current line from file */
2417: SPL_METHOD(SplFileObject, current)
2418: {
2419:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2420:
2421:   if (zend_parse_parameters_none() == FAILURE) {
2422:     return;
2423:   }
2424:
2425:   if(!intern->u.file.stream) {
2426:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2427:     return;
2428:   }
2429:
2430:   if (!intern->u.file.current_line && Z_ISUNDEF(intern->u.file.current_zval)) {
2431:     spl_filesystem_file_read_line(getThis(), intern, 1);
2432:   }
2433:   if (intern->u.file.current_line && (!SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_CSV) || Z_ISUNDEF(intern->u.file.current_zval))) {
2434:     RETURN_STRINGL(intern->u.file.current_line, intern->u.file.current_line_len);
2435:   } else if (!Z_ISUNDEF(intern->u.file.current_zval)) {
2436:     zval *value = &intern->u.file.current_zval;
2437:
2438:     ZVAL_DEREF(value);
2439:     ZVAL_COPY(return_value, value);
2440:     return;
2441:   }
2442:   RETURN_FALSE;
2443: } /* }}} */
2444:
2445: /* {{{ proto int SplFileObject::key()
2446:    Return line number */
2447: SPL_METHOD(SplFileObject, key)
2448: {
2449:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2450:
2451:   if (zend_parse_parameters_none() == FAILURE) {
2452:     return;
2453:   }
2454:
2455: /*  Do not read the next line to support correct counting with fgetc()
2456:   if (!intern->current_line) {
2457:     spl_filesystem_file_read_line(getThis(), intern, 1);
2458:   } */
2459:   RETURN_LONG(intern->u.file.current_line_num);
2460: } /* }}} */
2461:
2462: /* {{{ proto void SplFileObject::next()
2463:    Read next line */
2464: SPL_METHOD(SplFileObject, next)
2465: {
2466:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2467:
2468:   if (zend_parse_parameters_none() == FAILURE) {
2469:     return;
2470:   }
2471:
2472:   spl_filesystem_file_free_line(intern);
2473:   if (SPL_HAS_FLAG(intern->flags, SPL_FILE_OBJECT_READ_AHEAD)) {
2474:     spl_filesystem_file_read_line(getThis(), intern, 1);
2475:   }
2476:   intern->u.file.current_line_num++;
2477: } /* }}} */
2478:
2479: /* {{{ proto void SplFileObject::setFlags(int flags)
2480:    Set file handling flags */
2481: SPL_METHOD(SplFileObject, setFlags)
2482: {
2483:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2484:
2485:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &intern->flags) == FAILURE) {
2486:     return;
2487:   }
2488: } /* }}} */
2489:
2490: /* {{{ proto int SplFileObject::getFlags()
2491:    Get file handling flags */
2492: SPL_METHOD(SplFileObject, getFlags)
2493: {
2494:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2495:
2496:   if (zend_parse_parameters_none() == FAILURE) {
2497:     return;
2498:   }
2499:
2500:   RETURN_LONG(intern->flags & SPL_FILE_OBJECT_MASK);
2501: } /* }}} */
2502:
2503: /* {{{ proto void SplFileObject::setMaxLineLen(int max_len)
2504:    Set maximum line length */
2505: SPL_METHOD(SplFileObject, setMaxLineLen)
2506: {
2507:   zend_long max_len;
2508:
2509:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2510:
2511:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &max_len) == FAILURE) {
2512:     return;
2513:   }
2514:
2515:   if (max_len < 0) {
2516:     zend_throw_exception_ex(spl_ce_DomainException, 0, "Maximum line length must be greater than or equal zero");
2517:     return;
2518:   }
2519:
2520:   intern->u.file.max_line_len = max_len;
2521: } /* }}} */
2522:
2523: /* {{{ proto int SplFileObject::getMaxLineLen()
2524:    Get maximum line length */
2525: SPL_METHOD(SplFileObject, getMaxLineLen)
2526: {
2527:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2528:
2529:   if (zend_parse_parameters_none() == FAILURE) {
2530:     return;
2531:   }
2532:
2533:   RETURN_LONG((zend_long)intern->u.file.max_line_len);
2534: } /* }}} */
2535:
2536: /* {{{ proto bool SplFileObject::hasChildren()
2537:    Return false */
2538: SPL_METHOD(SplFileObject, hasChildren)
2539: {
2540:   if (zend_parse_parameters_none() == FAILURE) {
2541:     return;
2542:   }
2543:
2544:   RETURN_FALSE;
2545: } /* }}} */
2546:
2547: /* {{{ proto bool SplFileObject::getChildren()
2548:    Read NULL */
2549: SPL_METHOD(SplFileObject, getChildren)
2550: {
2551:   if (zend_parse_parameters_none() == FAILURE) {
2552:     return;
2553:   }
2554:   /* return NULL */
2555: } /* }}} */
2556:
2557: /* {{{ FileFunction */
2558: #define FileFunction(func_name) \
2559: SPL_METHOD(SplFileObject, func_name) \
2560: { \
2561:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis()); \
2562:   FileFunctionCall(func_name, ZEND_NUM_ARGS(), NULL); \
2563: }
2564: /* }}} */
2565:
2566: /* {{{ proto array SplFileObject::fgetcsv([string delimiter [, string enclosure [, escape = '\\']]])
2567:    Return current line as csv */
2568: SPL_METHOD(SplFileObject, fgetcsv)
2569: {
2570:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2571:   char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
2572:   char *delim = NULL, *enclo = NULL, *esc = NULL;
2573:   size_t d_len = 0, e_len = 0, esc_len = 0;
2574:
2575:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "|sss", &delim, &d_len, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2576:
2577:     if(!intern->u.file.stream) {
2578:       zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2579:       return;
2580:     }
2581:
2582:     switch(ZEND_NUM_ARGS())
2583:     {
2584:     case 3:
2585:       if (esc_len != 1) {
2586:         php_error_docref(NULL, E_WARNING, "escape must be a character");
2587:         RETURN_FALSE;
2588:       }
2589:       escape = esc[0];
2590:       /* no break */
2591:     case 2:
2592:       if (e_len != 1) {
2593:         php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2594:         RETURN_FALSE;
2595:       }
2596:       enclosure = enclo[0];
2597:       /* no break */
2598:     case 1:
2599:       if (d_len != 1) {
2600:         php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2601:         RETURN_FALSE;
2602:       }
2603:       delimiter = delim[0];
2604:       /* no break */
2605:     case 0:
2606:       break;
2607:     }
2608:     spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
2609:   }
2610: }
2611: /* }}} */
2612:
2613: /* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]]])
2614:    Output a field array as a CSV line */
2615: SPL_METHOD(SplFileObject, fputcsv)
2616: {
2617:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2618:   char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
2619:   char *delim = NULL, *enclo = NULL, *esc = NULL;
2620:   size_t d_len = 0, e_len = 0, esc_len = 0;
2621:   zend_long ret;
2622:   zval *fields = NULL;
2623:
2624:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "a|sss", &fields, &delim, &d_len, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2625:     switch(ZEND_NUM_ARGS())
2626:     {
2627:     case 4:
2628:       if (esc_len != 1) {
```

```
2629:         php_error_docref(NULL, E_WARNING, "escape must be a character");
2630:         RETURN_FALSE;
2631:       }
2632:       escape = esc[0];
2633:       /* no break */
2634:     case 3:
2635:       if (e_len != 1) {
2636:         php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2637:         RETURN_FALSE;
2638:       }
2639:       enclosure = enclo[0];
2640:       /* no break */
2641:     case 2:
2642:       if (d_len != 1) {
2643:         php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2644:         RETURN_FALSE;
2645:       }
2646:       delimiter = delim[0];
2647:       /* no break */
2648:     case 1:
2649:     case 0:
2650:       break;
2651:     }
2652:     ret = php_fputcsv(intern->u.file.stream, fields, delimiter, enclosure, escape);
2653:     RETURN_LONG(ret);
2654:   }
2655: }
2656: /* }}} */
2657:
2658: /* {{{ proto void SplFileObject::setCsvControl([string delimiter [, string enclosure [, string escape ]]])
2659:    Set the delimiter, enclosure and escape character used in fgetcsv */
2660: SPL_METHOD(SplFileObject, setCsvControl)
2661: {
2662:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2663:   char delimiter = ',', enclosure = '"', escape='\\';
2664:   char *delim = NULL, *enclo = NULL, *esc = NULL;
2665:   size_t d_len = 0, e_len = 0, esc_len = 0;
2666:
2667:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "|sss", &delim, &d_len, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2668:     switch(ZEND_NUM_ARGS())
2669:     {
2670:     case 3:
2671:       if (esc_len != 1) {
2672:         php_error_docref(NULL, E_WARNING, "escape must be a character");
2673:         RETURN_FALSE;
2674:       }
2675:       escape = esc[0];
2676:       /* no break */
2677:     case 2:
2678:       if (e_len != 1) {
2679:         php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2680:         RETURN_FALSE;
2681:       }
2682:       enclosure = enclo[0];
2683:       /* no break */
2684:     case 1:
2685:       if (d_len != 1) {
2686:         php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2687:         RETURN_FALSE;
2688:       }
2689:       delimiter = delim[0];
2690:       /* no break */
2691:     case 0:
2692:       break;
2693:     }
2694:     intern->u.file.delimiter = delimiter;
2695:     intern->u.file.enclosure = enclosure;
2696:     intern->u.file.escape    = escape;
2697:   }
2698: }
2699: /* }}} */
2700:
2701: /* {{{ proto array SplFileObject::getCsvControl()
2702:    Get the delimiter, enclosure and escape character used in fgetcsv */
2703: SPL_METHOD(SplFileObject, getCsvControl)
2704: {
2705:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2706:   char delimiter[2], enclosure[2], escape[2];
2707:
2708:   array_init(return_value);
2709:
2710:   delimiter[0] = intern->u.file.delimiter;
2711:   delimiter[1] = '\0';
2712:   enclosure[0] = intern->u.file.enclosure;
2713:   enclosure[1] = '\0';
2714:   escape[0] = intern->u.file.escape;
2715:   escape[1] = '\0';
2716:
2717:   add_next_index_string(return_value, delimiter);
2718:   add_next_index_string(return_value, enclosure);
2719:   add_next_index_string(return_value, escape);
2720: }
2721: /* }}} */
2722:
2723: /* {{{ proto bool SplFileObject::flock(int operation [, int &wouldblock])
2724:    Portable file locking */
2725: FileFunction(flock)
2726: /* }}} */
2727:
2728: /* {{{ proto bool SplFileObject::fflush()
2729:    Flush the file */
2730: SPL_METHOD(SplFileObject, fflush)
2731: {
2732:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2733:
2734:   if(!intern->u.file.stream) {
2735:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2736:     return;
2737:   }
2738:
2739:   RETURN_BOOL(!php_stream_flush(intern->u.file.stream));
2740: } /* }}} */
2741:
2742: /* {{{ proto int SplFileObject::ftell()
2743:    Return current file position */
2744: SPL_METHOD(SplFileObject, ftell)
2745: {
2746:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2747:   zend_long ret;
2748:
2749:   if(!intern->u.file.stream) {
2750:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2751:     return;
2752:   }
2753:
2754:   ret = php_stream_tell(intern->u.file.stream);
2755:
2756:   if (ret == -1) {
2757:     RETURN_FALSE;
2758:   } else {
2759:     RETURN_LONG(ret);
2760:   }
2761: } /* }}} */
2762:
2763: /* {{{ proto int SplFileObject::fseek(int pos [, int whence = SEEK_SET])
2764:    Return current file position */
2765: SPL_METHOD(SplFileObject, fseek)
2766: {
2767:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2768:   zend_long pos, whence = SEEK_SET;
2769:
2770:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l|l", &pos, &whence) == FAILURE) {
2771:     return;
2772:   }
2773:
2774:   if(!intern->u.file.stream) {
2775:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2776:     return;
2777:   }
2778:
2779:   spl_filesystem_file_free_line(intern);
2780:   RETURN_LONG(php_stream_seek(intern->u.file.stream, pos, (int)whence));
2781: } /* }}} */
2782:
2783: /* {{{ proto int SplFileObject::fgetc()
2784:    Get a character form the file */
2785: SPL_METHOD(SplFileObject, fgetc)
2786: {
2787:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2788:   char buf[2];
2789:   int result;
2790:
2791:   if(!intern->u.file.stream) {
2792:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2793:     return;
2794:   }
2795:
2796:   spl_filesystem_file_free_line(intern);
2797:
2798:   result = php_stream_getc(intern->u.file.stream);
2799:
2800:   if (result == EOF) {
2801:     RETVAL_FALSE;
2802:   } else {
2803:     if (result == '\n') {
2804:       intern->u.file.current_line_num++;
2805:     }
2806:     buf[0] = result;
2807:     buf[1] = '\0';
2808:
2809:     RETURN_STRINGL(buf, 1);
2810:   }
2811: } /* }}} */
2812:
2813: /* {{{ proto string SplFileObject::fgetss([string allowable_tags])
2814:    Get a line from file pointer and strip HTML tags */
2815: SPL_METHOD(SplFileObject, fgetss)
2816: {
2817:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2818:   zval arg2;
2819:
2820:   if(!intern->u.file.stream) {
2821:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2822:     return;
2823:   }
2824:
2825:   if (intern->u.file.max_line_len > 0) {
2826:     ZVAL_LONG(&arg2, intern->u.file.max_line_len);
2827:   } else {
2828:     ZVAL_LONG(&arg2, 1024);
2829:   }
2830:
2831:   spl_filesystem_file_free_line(intern);
2832:   intern->u.file.current_line_num++;
2833:
2834:   FileFunctionCall(fgetss, ZEND_NUM_ARGS(), &arg2);
2835: } /* }}} */
2836:
2837: /* {{{ proto int SplFileObject::fpassthru()
2838:    Output all remaining data from a file pointer */
2839: SPL_METHOD(SplFileObject, fpassthru)
2840: {
2841:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2842:
2843:   if(!intern->u.file.stream) {
2844:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2845:     return;
2846:   }
2847:
2848:   RETURN_LONG(php_stream_passthru(intern->u.file.stream));
2849: } /* }}} */
2850:
2851: /* {{{ proto bool SplFileObject::fscanf(string format [, string ...])
2852:    Implements a mostly ANSI compatible fscanf() */
2853: SPL_METHOD(SplFileObject, fscanf)
2854: {
2855:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2856:
2857:   if(!intern->u.file.stream) {
2858:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2859:     return;
2860:   }
2861:
2862:   spl_filesystem_file_free_line(intern);
2863:   intern->u.file.current_line_num++;
2864:
2865:   FileFunctionCall(fscanf, ZEND_NUM_ARGS(), NULL);
2866: }
2867: /* }}} */
2868:
2869: /* {{{ proto mixed SplFileObject::fwrite(string str [, int length])
2870:    Binary-safe file write */
2871: SPL_METHOD(SplFileObject, fwrite)
2872: {
2873:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2874:   char *str;
2875:   size_t str_len;
2876:   zend_long length = 0;
2877:
2878:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "s|l", &str, &str_len, &length) == FAILURE) {
2879:     return;
2880:   }
2881:
2882:   if(!intern->u.file.stream) {
2883:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2884:     return;
2885:   }
2886:
2887:   if (ZEND_NUM_ARGS() > 1) {
2888:     if (length >= 0) {
2889:       str_len = MIN((size_t)length, str_len);
2890:     } else {
2891:       /* Negative length given, nothing to write */
2892:       str_len = 0;
2893:     }
2894:   }
2895:   if (!str_len) {
2896:     RETURN_LONG(0);
2897:   }
2898:
2899:   RETURN_LONG(php_stream_write(intern->u.file.stream, str, str_len));
2900: } /* }}} */
2901:
2902: SPL_METHOD(SplFileObject, fread)
2903: {
2904:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2905:   zend_long length = 0;
2906:
2907:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &length) == FAILURE) {
2908:     return;
2909:   }
2910:
2911:   if(!intern->u.file.stream) {
2912:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2913:     return;
2914:   }
2915:
2916:   if (length <= 0) {
2917:     php_error_docref(NULL, E_WARNING, "Length parameter must be greater than 0");
2918:     RETURN_FALSE;
2919:   }
2920:
2921:   ZVAL_NEW_STR(return_value, zend_string_alloc(length, 0));
2922:   Z_STRLEN_P(return_value) = php_stream_read(intern->u.file.stream, Z_STRVAL_P(return_value), length);
2923:
2924:   /* needed because recv/read/gzread doesnt put a null at the end*/
2925:   Z_STRVAL_P(return_value)[Z_STRLEN_P(return_value)] = 0;
2926: }
2927:
2928: /* {{{ proto bool SplFileObject::fstat()
2929:    Stat() on a filehandle */
2930: FileFunction(fstat)
2931: /* }}} */
2932:
2933: /* {{{ proto bool SplFileObject::ftruncate(int size)
2934:    Truncate file to 'size' length */
2935: SPL_METHOD(SplFileObject, ftruncate)
2936: {
2937:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2938:   zend_long size;
2939:
2940:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &size) == FAILURE) {
2941:     return;
2942:   }
2943:
2944:   if(!intern->u.file.stream) {
2945:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2946:     return;
2947:   }
2948:
2949:   if (!php_stream_truncate_supported(intern->u.file.stream)) {
2950:     zend_throw_exception_ex(spl_ce_LogicException, 0, "Can't truncate file %s", intern->file_name);
2951:     RETURN_FALSE;
2952:   }
2953:
2954:   RETURN_BOOL(0 == php_stream_truncate_set_size(intern->u.file.stream, size));
2955: } /* }}} */
2956:
2957: /* {{{ proto void SplFileObject::seek(int line_pos)
2958:    Seek to specified line */
2959: SPL_METHOD(SplFileObject, seek)
2960: {
2961:   spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(getThis());
2962:   zend_long line_pos;
2963:
2964:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &line_pos) == FAILURE) {
2965:     return;
2966:   }
2967:   if(!intern->u.file.stream) {
2968:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2969:     return;
2970:   }
2971:
2972:   if (line_pos < 0) {
2973:     zend_throw_exception_ex(spl_ce_LogicException, 0, "Can't seek file %s to negative line " ZEND_LONG_FMT, intern->file_name, line_pos);
2974:     RETURN_FALSE;
2975:   }
2976:
2977:   spl_filesystem_file_rewind(getThis(), intern);
2978:
2979:   while(intern->u.file.current_line_num < line_pos) {
2980:     if (spl_filesystem_file_read_line(getThis(), intern, 1) == FAILURE) {
2981:       break;
2982:     }
2983:   }
2984: } /* }}} */
2985:
2986: /* {{{ Function/Class/Method definitions */
2987: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object___construct, 0, 0, 1)
2988:   ZEND_ARG_INFO(0, file_name)
2989:   ZEND_ARG_INFO(0, open_mode)
2990:   ZEND_ARG_INFO(0, use_include_path)
2991:   ZEND_ARG_INFO(0, context)
2992: ZEND_END_ARG_INFO()
2993:
2994: ZEND_BEGIN_ARG_INFO(arginfo_file_object_setFlags, 0)
2995:   ZEND_ARG_INFO(0, flags)
2996: ZEND_END_ARG_INFO()
2997:
2998: ZEND_BEGIN_ARG_INFO(arginfo_file_object_setMaxLineLen, 0)
2999:   ZEND_ARG_INFO(0, max_len)
3000: ZEND_END_ARG_INFO()
3001:
3002: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fgetcsv, 0, 0, 0)
3003:   ZEND_ARG_INFO(0, delimiter)
3004:   ZEND_ARG_INFO(0, enclosure)
```

```
3005:   ZEND_ARG_INFO(0, escape)
3006: ZEND_END_ARG_INFO()
3007:
3008: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fputcsv, 0, 0, 1)
3009:   ZEND_ARG_INFO(0, fields)
3010:   ZEND_ARG_INFO(0, delimiter)
3011:   ZEND_ARG_INFO(0, enclosure)
3012:   ZEND_ARG_INFO(0, escape)
3013: ZEND_END_ARG_INFO()
3014:
3015: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_flock, 0, 0, 1)
3016:   ZEND_ARG_INFO(0, operation)
3017:   ZEND_ARG_INFO(1, wouldblock)
3018: ZEND_END_ARG_INFO()
3019:
3020: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fseek, 0, 0, 1)
3021:   ZEND_ARG_INFO(0, pos)
3022:   ZEND_ARG_INFO(0, whence)
3023: ZEND_END_ARG_INFO()
3024:
3025: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fgetss, 0, 0, 0)
3026:   ZEND_ARG_INFO(0, allowable_tags)
3027: ZEND_END_ARG_INFO()
3028:
3029: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fscanf, 0, 0, 1)
3030:   ZEND_ARG_INFO(0, format)
3031:   ZEND_ARG_VARIADIC_INFO(1, vars)
3032: ZEND_END_ARG_INFO()
3033:
3034: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fwrite, 0, 0, 1)
3035:   ZEND_ARG_INFO(0, str)
3036:   ZEND_ARG_INFO(0, length)
3037: ZEND_END_ARG_INFO()
3038:
3039: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fread, 0, 0, 1)
3040:   ZEND_ARG_INFO(0, length)
3041: ZEND_END_ARG_INFO()
3042:
3043: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_ftruncate, 0, 0, 1)
3044:   ZEND_ARG_INFO(0, size)
3045: ZEND_END_ARG_INFO()
3046:
3047: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_seek, 0, 0, 1)
3048:   ZEND_ARG_INFO(0, line_pos)
3049: ZEND_END_ARG_INFO()
3050:
3051: static const zend_function_entry spl_SplFileObject_functions[] = {
3052:   SPL_ME(SplFileObject, __construct,    arginfo_file_object___construct,   ZEND_ACC_PUBLIC)
3053:   SPL_ME(SplFileObject, rewind,         arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3054:   SPL_ME(SplFileObject, eof,            arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3055:   SPL_ME(SplFileObject, valid,          arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3056:   SPL_ME(SplFileObject, fgets,          arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3057:   SPL_ME(SplFileObject, fgetcsv,        arginfo_file_object_fgetcsv,       ZEND_ACC_PUBLIC)
3058:   SPL_ME(SplFileObject, fputcsv,        arginfo_file_object_fputcsv,       ZEND_ACC_PUBLIC)
3059:   SPL_ME(SplFileObject, setCsvControl,  arginfo_file_object_fgetcsv,       ZEND_ACC_PUBLIC)
3060:   SPL_ME(SplFileObject, getCsvControl,  arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3061:   SPL_ME(SplFileObject, flock,          arginfo_file_object_flock,         ZEND_ACC_PUBLIC)
3062:   SPL_ME(SplFileObject, fflush,         arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3063:   SPL_ME(SplFileObject, ftell,          arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3064:   SPL_ME(SplFileObject, fseek,          arginfo_file_object_fseek,         ZEND_ACC_PUBLIC)
3065:   SPL_ME(SplFileObject, fgetc,          arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3066:   SPL_ME(SplFileObject, fpassthru,      arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3067:   SPL_ME(SplFileObject, fgetss,         arginfo_file_object_fgetss,        ZEND_ACC_PUBLIC)
3068:   SPL_ME(SplFileObject, fscanf,         arginfo_file_object_fscanf,        ZEND_ACC_PUBLIC)
3069:   SPL_ME(SplFileObject, fwrite,         arginfo_file_object_fwrite,        ZEND_ACC_PUBLIC)
3070:   SPL_ME(SplFileObject, fread,          arginfo_file_object_fread,         ZEND_ACC_PUBLIC)
3071:   SPL_ME(SplFileObject, fstat,          arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3072:   SPL_ME(SplFileObject, ftruncate,      arginfo_file_object_ftruncate,     ZEND_ACC_PUBLIC)
3073:   SPL_ME(SplFileObject, current,        arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3074:   SPL_ME(SplFileObject, key,            arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3075:   SPL_ME(SplFileObject, next,           arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3076:   SPL_ME(SplFileObject, setFlags,       arginfo_file_object_setFlags,      ZEND_ACC_PUBLIC)
3077:   SPL_ME(SplFileObject, getFlags,       arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3078:   SPL_ME(SplFileObject, setMaxLineLen,  arginfo_file_object_setMaxLineLen, ZEND_ACC_PUBLIC)
3079:   SPL_ME(SplFileObject, getMaxLineLen,  arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3080:   SPL_ME(SplFileObject, hasChildren,    arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3081:   SPL_ME(SplFileObject, getChildren,    arginfo_splfileinfo_void,          ZEND_ACC_PUBLIC)
3082:   SPL_ME(SplFileObject, seek,           arginfo_file_object_seek,          ZEND_ACC_PUBLIC)
3083:   /* mappings */
3084:   SPL_MA(SplFileObject, getCurrentLine, SplFileObject, fgets,      arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3085:   SPL_MA(SplFileObject, __toString,     SplFileObject, current,   arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3086:   PHP_FE_END
3087: };
3088:
3089: ZEND_BEGIN_ARG_INFO_EX(arginfo_temp_file_object___construct, 0, 0, 0)
3090:   ZEND_ARG_INFO(0, max_memory)
3091: ZEND_END_ARG_INFO()
3092:
3093: static const zend_function_entry spl_SplTempFileObject_functions[] = {
3094:   SPL_ME(SplTempFileObject, __construct, arginfo_temp_file_object___construct,  ZEND_ACC_PUBLIC)
3095:   PHP_FE_END
3096: };
3097: /* }}} */
3098:
3099: /* {{{ PHP_MINIT_FUNCTION(spl_directory)
3100:  */
3101: PHP_MINIT_FUNCTION(spl_directory)
3102: {
3103:   REGISTER_SPL_STD_CLASS_EX(SplFileInfo, spl_filesystem_object_new, spl_SplFileInfo_functions);
3104:   memcpy(&spl_filesystem_object_handlers, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
3105:   spl_filesystem_object_handlers.offset = XtOffsetOf(spl_filesystem_object, std);
3106:   spl_filesystem_object_handlers.clone_obj = spl_filesystem_object_clone;
3107:   spl_filesystem_object_handlers.cast_object = spl_filesystem_object_cast;
3108:   spl_filesystem_object_handlers.get_debug_info  = spl_filesystem_object_get_debug_info;
3109:   spl_filesystem_object_handlers.dtor_obj = spl_filesystem_object_destroy_object;
3110:   spl_filesystem_object_handlers.free_obj = spl_filesystem_object_free_storage;
3111:   spl_ce_SplFileInfo->serialize = zend_class_serialize_deny;
3112:   spl_ce_SplFileInfo->unserialize = zend_class_unserialize_deny;
3113:
3114:
3115:   REGISTER_SPL_SUB_CLASS_EX(DirectoryIterator, SplFileInfo, spl_filesystem_object_new, spl_DirectoryIterator_functions);
3116:   zend_class_implements(spl_ce_DirectoryIterator, 1, zend_ce_iterator);
3117:   REGISTER_SPL_IMPLEMENTS(DirectoryIterator, SeekableIterator);
3118:
3119:   spl_ce_DirectoryIterator->get_iterator = spl_filesystem_dir_get_iterator;
3120:
3121:   REGISTER_SPL_SUB_CLASS_EX(FilesystemIterator, DirectoryIterator, spl_filesystem_object_new, spl_FilesystemIterator_functions);
3122:
3123:   REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_MODE_MASK",   SPL_FILE_DIR_CURRENT_MODE_MASK);
3124:   REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_PATHNAME", SPL_FILE_DIR_CURRENT_AS_PATHNAME);
3125:   REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_FILEINFO", SPL_FILE_DIR_CURRENT_AS_FILEINFO);
3126:   REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_SELF",     SPL_FILE_DIR_CURRENT_AS_SELF);
3127:   REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_MODE_MASK",      SPL_FILE_DIR_KEY_MODE_MASK);
3128:   REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_AS_PATHNAME",    SPL_FILE_DIR_KEY_AS_PATHNAME);
3129:   REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "FOLLOW_SYMLINKS",    SPL_FILE_DIR_FOLLOW_SYMLINKS);
3130:   REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_AS_FILENAME",    SPL_FILE_DIR_KEY_AS_FILENAME);
3131:   REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "NEW_CURRENT_AND_KEY", SPL_FILE_DIR_KEY_AS_FILENAME|SPL_FILE_DIR_CURRENT_AS_FILEINFO);
3132:   REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "OTHER_MODE_MASK",     SPL_FILE_DIR_OTHERS_MASK);
3133:   REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "SKIP_DOTS",          SPL_FILE_DIR_SKIPDOTS);
3134:   REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "UNIX_PATHS",         SPL_FILE_DIR_UNIXPATHS);
3135:
3136:   spl_ce_FilesystemIterator->get_iterator = spl_filesystem_tree_get_iterator;
3137:
3138:   REGISTER_SPL_SUB_CLASS_EX(RecursiveDirectoryIterator, FilesystemIterator, spl_filesystem_object_new, spl_RecursiveDirectoryIterator_functions);
3139:   REGISTER_SPL_IMPLEMENTS(RecursiveDirectoryIterator, RecursiveIterator);
3140:
3141:   memcpy(&spl_filesystem_object_check_handlers, &spl_filesystem_object_handlers, sizeof(zend_object_handlers));
3142:   spl_filesystem_object_check_handlers.get_method = spl_filesystem_object_get_method_check;
3143:
3144: #ifdef HAVE_GLOB
3145:   REGISTER_SPL_SUB_CLASS_EX(GlobIterator, FilesystemIterator, spl_filesystem_object_new_check, spl_GlobIterator_functions);
3146:   REGISTER_SPL_IMPLEMENTS(GlobIterator, Countable);
3147: #endif
3148:
3149:   REGISTER_SPL_SUB_CLASS_EX(SplFileObject, SplFileInfo, spl_filesystem_object_new_check, spl_SplFileObject_functions);
3150:   REGISTER_SPL_IMPLEMENTS(SplFileObject, RecursiveIterator);
3151:   REGISTER_SPL_IMPLEMENTS(SplFileObject, SeekableIterator);
3152:
3153:   REGISTER_SPL_CLASS_CONST_LONG(SplFileObject, "DROP_NEW_LINE", SPL_FILE_OBJECT_DROP_NEW_LINE);
3154:   REGISTER_SPL_CLASS_CONST_LONG(SplFileObject, "READ_AHEAD",    SPL_FILE_OBJECT_READ_AHEAD);
3155:   REGISTER_SPL_CLASS_CONST_LONG(SplFileObject, "SKIP_EMPTY",    SPL_FILE_OBJECT_SKIP_EMPTY);
3156:   REGISTER_SPL_CLASS_CONST_LONG(SplFileObject, "READ_CSV",      SPL_FILE_OBJECT_READ_CSV);
3157:
3158:   REGISTER_SPL_SUB_CLASS_EX(SplTempFileObject, SplFileObject, spl_filesystem_object_new_check, spl_SplTempFileObject_functions);
3159:   return SUCCESS;
3160: }
3161: /* }}} */
3162:
3163: /*
3164:  * Local variables:
3165:  * tab-width: 4
3166:  * c-basic-offset: 4
3167:  * End:
3168:  * vim600: noet sw=4 ts=4 fdm=marker
3169:  * vim<600: noet sw=4 ts=4
3170:  */
```

```
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_ITERATORS_H
#define SPL_ITERATORS_H

#include "php.h"
#include "php_spl.h"
#if HAVE_PCRE || HAVE_BUNDLED_PCRE
#include "ext/pcre/php_pcre.h"
#endif

#define spl_ce_Traversable   zend_ce_traversable
#define spl_ce_Iterator      zend_ce_iterator
#define spl_ce_Aggregate     zend_ce_aggregate
#define spl_ce_ArrayAccess   zend_ce_arrayaccess
#define spl_ce_Serializable  zend_ce_serializable
#define spl_ce_Countable     zend_ce_countable

extern PHPAPI zend_class_entry *spl_ce_RecursiveIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveIteratorIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveTreeIterator;
extern PHPAPI zend_class_entry *spl_ce_FilterIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveFilterIterator;
extern PHPAPI zend_class_entry *spl_ce_ParentIterator;
extern PHPAPI zend_class_entry *spl_ce_SeekableIterator;
extern PHPAPI zend_class_entry *spl_ce_LimitIterator;
extern PHPAPI zend_class_entry *spl_ce_CachingIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveCachingIterator;
extern PHPAPI zend_class_entry *spl_ce_OuterIterator;
extern PHPAPI zend_class_entry *spl_ce_IteratorIterator;
extern PHPAPI zend_class_entry *spl_ce_NoRewindIterator;
extern PHPAPI zend_class_entry *spl_ce_InfiniteIterator;
extern PHPAPI zend_class_entry *spl_ce_EmptyIterator;
extern PHPAPI zend_class_entry *spl_ce_AppendIterator;
extern PHPAPI zend_class_entry *spl_ce_RegexIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveRegexIterator;
extern PHPAPI zend_class_entry *spl_ce_CallbackFilterIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveCallbackFilterIterator;

PHP_MINIT_FUNCTION(spl_iterators);

PHP_FUNCTION(iterator_to_array);
PHP_FUNCTION(iterator_count);
PHP_FUNCTION(iterator_apply);

typedef enum {
  DIT_Default = 0,
  DIT_FilterIterator = DIT_Default,
  DIT_RecursiveFilterIterator = DIT_Default,
  DIT_ParentIterator = DIT_Default,
  DIT_LimitIterator,
  DIT_CachingIterator,
  DIT_RecursiveCachingIterator,
  DIT_IteratorIterator,
  DIT_NoRewindIterator,
  DIT_InfiniteIterator,
  DIT_AppendIterator,
#if HAVE_PCRE || HAVE_BUNDLED_PCRE
  DIT_RegexIterator,
  DIT_RecursiveRegexIterator,
#endif
  DIT_CallbackFilterIterator,
  DIT_RecursiveCallbackFilterIterator,
  DIT_Unknown = ~0
} dual_it_type;

typedef enum {
  RIT_Default = 0,
  RIT_RecursiveIteratorIterator = RIT_Default,
  RIT_RecursiveTreeIterator,
  RIT_Unknow = ~0
} recursive_it_it_type;

enum {
  /* public */
  CIT_CALL_TOSTRING       = 0x00000001,
  CIT_TOSTRING_USE_KEY    = 0x00000002,
  CIT_TOSTRING_USE_CURRENT = 0x00000004,
  CIT_TOSTRING_USE_INNER  = 0x00000008,
  CIT_CATCH_GET_CHILD     = 0x00000010,
  CIT_FULL_CACHE          = 0x00000100,
  CIT_PUBLIC              = 0x0000FFFF,
  /* private */
  CIT_VALID               = 0x00010000,
  CIT_HAS_CHILDREN        = 0x00020000
};

enum {
  /* public */
  REGIT_USE_KEY           = 0x00000001,
  REGIT_INVERTED          = 0x00000002
};

typedef enum {
  REGIT_MODE_MATCH,
  REGIT_MODE_GET_MATCH,
  REGIT_MODE_ALL_MATCHES,
  REGIT_MODE_SPLIT,
  REGIT_MODE_REPLACE,
  REGIT_MODE_MAX
} regex_mode;

typedef struct _spl_cbfilter_it_intern {
  zend_fcall_info       fci;
  zend_fcall_info_cache fcc;
  zend_object           *object;
} _spl_cbfilter_it_intern;

typedef struct _spl_dual_it_object {
  struct {
    zval              zobject;
    zend_class_entry  *ce;
    zend_object       *object;
    zend_object_iterator *iterator;
  } inner;
  struct {
    zval              data;
    zval              key;
    zend_long         pos;
  } current;
  dual_it_type            dit_type;
  union {
    struct {
      zend_long           offset;
      zend_long           count;
    } limit;
    struct {
      zend_long           flags; /* CIT_* */
      zval            zstr;
      zval            zchildren;
      zval            zcache;
    } caching;
    struct {
      zval                zarrayit;
      zend_object_iterator *iterator;
    } append;
#if HAVE_PCRE || HAVE_BUNDLED_PCRE
    struct {
      zend_long       flags;
      zend_long       preg_flags;
      pcre_cache_entry *pce;
      zend_string     *regex;
      regex_mode      mode;
      int             use_flags;
    } regex;
#endif
    _spl_cbfilter_it_intern *cbfilter;
  } u;
  zend_object             std;
} spl_dual_it_object;

static inline spl_dual_it_object *spl_dual_it_from_obj(zend_object *obj) /* {{{ */ {
  return (spl_dual_it_object*)((char*)(obj) - XtOffsetOf(spl_dual_it_object, std));
} /* }}} */

#define Z_SPLDUAL_IT_P(zv)  spl_dual_it_from_obj(Z_OBJ_P((zv)))

typedef int (*spl_iterator_apply_func_t)(zend_object_iterator *iter, void *puser);

PHPAPI int spl_iterator_apply(zval *obj, spl_iterator_apply_func_t apply_func, void *puser);

#endif /* SPL_ITERATORS_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "ext/standard/php_var.h"
#include "zend_smart_str.h"
#include "zend_interfaces.h"
#include "zend_exceptions.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_iterators.h"
#include "spl_array.h"
#include "spl_exceptions.h"

zend_object_handlers spl_handler_ArrayObject;
PHPAPI zend_class_entry  *spl_ce_ArrayObject;

zend_object_handlers spl_handler_ArrayIterator;
PHPAPI zend_class_entry  *spl_ce_ArrayIterator;
PHPAPI zend_class_entry  *spl_ce_RecursiveArrayIterator;

#define SPL_ARRAY_STD_PROP_LIST      0x00000001
#define SPL_ARRAY_ARRAY_AS_PROPS     0x00000002
#define SPL_ARRAY_CHILD_ARRAYS_ONLY  0x00000004
#define SPL_ARRAY_OVERLOADED_REWIND  0x00010000
#define SPL_ARRAY_OVERLOADED_VALID   0x00020000
#define SPL_ARRAY_OVERLOADED_KEY     0x00040000
#define SPL_ARRAY_OVERLOADED_CURRENT 0x00080000
#define SPL_ARRAY_OVERLOADED_NEXT    0x00100000
#define SPL_ARRAY_IS_SELF            0x01000000
#define SPL_ARRAY_USE_OTHER          0x02000000
#define SPL_ARRAY_INT_MASK           0xFFFF0000
#define SPL_ARRAY_CLONE_MASK         0x0100FFFF

#define SPL_ARRAY_METHOD_NO_ARG       0
#define SPL_ARRAY_METHOD_USE_ARG      1
#define SPL_ARRAY_METHOD_MAY_USER_ARG 2

typedef struct _spl_array_object {
	zval             array;
	uint32_t         ht_iter;
	int              ar_flags;
	unsigned char    nApplyCount;
	zend_function    *fptr_offset_get;
	zend_function    *fptr_offset_set;
	zend_function    *fptr_offset_has;
	zend_function    *fptr_offset_del;
	zend_function    *fptr_count;
	zend_class_entry* ce_get_iterator;
	zend_object      std;
} spl_array_object;

static inline spl_array_object *spl_array_from_obj(zend_object *obj) /* {{{ */ {
	return (spl_array_object*)((char*)(obj) - XtOffsetOf(spl_array_object, std));
}
/* }}} */

#define Z_SPLARRAY_P(zv)  spl_array_from_obj(Z_OBJ_P((zv)))

static inline HashTable **spl_array_get_hash_table_ptr(spl_array_object* intern) { /* {{{ */
	//??? TODO: Delay duplication for arrays, only duplicate for write operations
	if (intern->ar_flags & SPL_ARRAY_IS_SELF) {
		if (!intern->std.properties) {
			rebuild_object_properties(&intern->std);
		}
		return &intern->std.properties;
	} else if (intern->ar_flags & SPL_ARRAY_USE_OTHER) {
		spl_array_object *other = Z_SPLARRAY_P(&intern->array);
		return spl_array_get_hash_table_ptr(other);
	} else if (Z_TYPE(intern->array) == IS_ARRAY) {
		return &Z_ARRVAL(intern->array);
	} else {
		zend_object *obj = Z_OBJ(intern->array);
		if (!obj->properties) {
			rebuild_object_properties(obj);
		} else if (GC_REFCOUNT(obj->properties) > 1) {
			if (EXPECTED(!(GC_FLAGS(obj->properties) & IS_ARRAY_IMMUTABLE))) {
				GC_DELREF(obj->properties);
			}
			obj->properties = zend_array_dup(obj->properties);
		}
		return &obj->properties;
	}
}
/* }}} */

static inline HashTable *spl_array_get_hash_table(spl_array_object* intern) { /* {{{ */
	return *spl_array_get_hash_table_ptr(intern);
}
/* }}} */

static inline void spl_array_replace_hash_table(spl_array_object* intern, HashTable *ht) { /* {{{ */
	HashTable **ht_ptr = spl_array_get_hash_table_ptr(intern);
	zend_array_destroy(*ht_ptr);
	*ht_ptr = ht;
}
/* }}} */

static inline zend_bool spl_array_is_object(spl_array_object *intern) /* {{{ */
{
	while (intern->ar_flags & SPL_ARRAY_USE_OTHER) {
		intern = Z_SPLARRAY_P(&intern->array);
	}
	return (intern->ar_flags & SPL_ARRAY_IS_SELF) || Z_TYPE(intern->array) == IS_OBJECT;
}
/* }}} */

static int spl_array_skip_protected(spl_array_object *intern, HashTable *aht);

static zend_never_inline void spl_array_create_ht_iter(HashTable *ht, spl_array_object* intern) /* {{{ */
{
	intern->ht_iter = zend_hash_iterator_add(ht, ht->nInternalPointer);
	zend_hash_internal_pointer_reset_ex(ht, &EG(ht_iterators)[intern->ht_iter].pos);
	spl_array_skip_protected(intern, ht);
}
/* }}} */

static zend_always_inline uint32_t *spl_array_get_pos_ptr(HashTable *ht, spl_array_object* intern) /* {{{ */
{
	if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
		spl_array_create_ht_iter(ht, intern);
	}
	return &EG(ht_iterators)[intern->ht_iter].pos;
}
/* }}} */

/* {{{ spl_array_object_free_storage */
static void spl_array_object_free_storage(zend_object *object)
{
	spl_array_object *intern = spl_array_from_obj(object);

	if (intern->ht_iter != (uint32_t) -1) {
		zend_hash_iterator_del(intern->ht_iter);
	}

	zend_object_std_dtor(&intern->std);

	zval_ptr_dtor(&intern->array);
}
/* }}} */

zend_object_iterator *spl_array_get_iterator(zend_class_entry *ce, zval *object, int by_ref);

/* {{{ spl_array_object_new_ex */
static zend_object *spl_array_object_new_ex(zend_class_entry *class_type, zval *orig, int clone_orig)
{
	spl_array_object *intern;
	zend_class_entry *parent = class_type;
	int inherited = 0;

	intern = zend_object_alloc(sizeof(spl_array_object), parent);

	zend_object_std_init(&intern->std, class_type);
	object_properties_init(&intern->std, class_type);

	intern->ar_flags = 0;
	intern->ce_get_iterator = spl_ce_ArrayIterator;
	if (orig) {
		spl_array_object *other = Z_SPLARRAY_P(orig);

		intern->ar_flags &= ~ SPL_ARRAY_CLONE_MASK;
		intern->ar_flags |= (other->ar_flags & SPL_ARRAY_CLONE_MASK);
		intern->ce_get_iterator = other->ce_get_iterator;
		if (clone_orig) {
			if (other->ar_flags & SPL_ARRAY_IS_SELF) {
				ZVAL_UNDEF(&intern->array);
			} else if (Z_OBJ_HT_P(orig) == &spl_handler_ArrayObject) {
				ZVAL_ARR(&intern->array,
					zend_array_dup(spl_array_get_hash_table(other)));
			} else {
				ZEND_ASSERT(Z_OBJ_HT_P(orig) == &spl_handler_ArrayIterator);
				ZVAL_COPY(&intern->array, orig);
				intern->ar_flags |= SPL_ARRAY_USE_OTHER;
			}
		} else {
			ZVAL_COPY(&intern->array, orig);
			intern->ar_flags |= SPL_ARRAY_USE_OTHER;
		}
	} else {
		array_init(&intern->array);
	}

	while (parent) {
		if (parent == spl_ce_ArrayIterator || parent == spl_ce_RecursiveArrayIterator) {
			intern->std.handlers = &spl_handler_ArrayIterator;
			class_type->get_iterator = spl_array_get_iterator;
			break;
		} else if (parent == spl_ce_ArrayObject) {
			intern->std.handlers = &spl_handler_ArrayObject;
			break;
		}
		parent = parent->parent;
		inherited = 1;
	}
	if (!parent) { /* this must never happen */
		php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error, Class is not child of ArrayObject or ArrayIterator");
	}
	if (inherited) {
		intern->fptr_offset_get = zend_hash_str_find_ptr(&class_type->function_table, "offsetget", sizeof("offsetget") - 1);
		if (intern->fptr_offset_get->common.scope == parent) {
			intern->fptr_offset_get = NULL;
		}
		intern->fptr_offset_set = zend_hash_str_find_ptr(&class_type->function_table, "offsetset", sizeof("offsetset") - 1);
		if (intern->fptr_offset_set->common.scope == parent) {
			intern->fptr_offset_set = NULL;
		}
		intern->fptr_offset_has = zend_hash_str_find_ptr(&class_type->function_table, "offsetexists", sizeof("offsetexists") - 1);
		if (intern->fptr_offset_has->common.scope == parent) {
			intern->fptr_offset_has = NULL;
		}
		intern->fptr_offset_del = zend_hash_str_find_ptr(&class_type->function_table, "offsetunset",  sizeof("offsetunset") - 1);
		if (intern->fptr_offset_del->common.scope == parent) {
			intern->fptr_offset_del = NULL;
		}
		intern->fptr_count = zend_hash_str_find_ptr(&class_type->function_table, "count", sizeof("count") - 1);
		if (intern->fptr_count->common.scope == parent) {
			intern->fptr_count = NULL;
		}
	}
	/* Cache iterator functions if ArrayIterator or derived. Check current's */
	/* cache since only current is always required */
	if (intern->std.handlers == &spl_handler_ArrayIterator) {
		if (!class_type->iterator_funcs.zf_current) {
			class_type->iterator_funcs.zf_rewind = zend_hash_str_find_ptr(&class_type->function_table, "rewind", sizeof("rewind") - 1);
			class_type->iterator_funcs.zf_valid = zend_hash_str_find_ptr(&class_type->function_table, "valid", sizeof("valid") - 1);
			class_type->iterator_funcs.zf_key = zend_hash_str_find_ptr(&class_type->function_table, "key", sizeof("key") - 1);
			class_type->iterator_funcs.zf_current = zend_hash_str_find_ptr(&class_type->function_table, "current", sizeof("current") - 1);
			class_type->iterator_funcs.zf_next = zend_hash_str_find_ptr(&class_type->function_table, "next", sizeof("next") - 1);
		}
		if (inherited) {
			if (class_type->iterator_funcs.zf_rewind->common.scope  != parent) intern->ar_flags |= SPL_ARRAY_OVERLOADED_REWIND;
			if (class_type->iterator_funcs.zf_valid->common.scope   != parent) intern->ar_flags |= SPL_ARRAY_OVERLOADED_VALID;
			if (class_type->iterator_funcs.zf_key->common.scope     != parent) intern->ar_flags |= SPL_ARRAY_OVERLOADED_KEY;
			if (class_type->iterator_funcs.zf_current->common.scope != parent) intern->ar_flags |= SPL_ARRAY_OVERLOADED_CURRENT;
			if (class_type->iterator_funcs.zf_next->common.scope    != parent) intern->ar_flags |= SPL_ARRAY_OVERLOADED_NEXT;
		}
	}

	intern->ht_iter = (uint32_t)-1;
	return &intern->std;
}
/* }}} */

/* {{{ spl_array_object_new */
static zend_object *spl_array_object_new(zend_class_entry *class_type)
{
	return spl_array_object_new_ex(class_type, NULL, 0);
}
/* }}} */

/* {{{ spl_array_object_clone */
static zend_object *spl_array_object_clone(zval *zobject)
{
	zend_object *old_object;
	zend_object *new_object;

	old_object = Z_OBJ_P(zobject);
	new_object = spl_array_object_new_ex(old_object->ce, zobject, 1);

	zend_objects_clone_members(new_object, old_object);

	return new_object;
}
/* }}} */

static zval *spl_array_get_dimension_ptr(int check_inherited, spl_array_object *intern, zval *offset, int type) /* {{{ */
{
	zval *retval;
	zend_long index;
	zend_string *offset_key;
	HashTable *ht = spl_array_get_hash_table(intern);

	if (!offset || Z_ISUNDEF_P(offset) || !ht) {
		return &EG(uninitialized_zval);
	}

	if ((type == BP_VAR_W || type == BP_VAR_RW) && intern->nApplyCount > 0) {
		zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
		return &EG(error_zval);
	}

try_again:
	switch (Z_TYPE_P(offset)) {
	case IS_NULL:
		offset_key = ZSTR_EMPTY_ALLOC();
		goto fetch_dim_string;
	case IS_STRING:
		offset_key = Z_STR_P(offset);
fetch_dim_string:
		retval = zend_symtable_find(ht, offset_key);
		if (retval) {
			if (Z_TYPE_P(retval) == IS_INDIRECT) {
				retval = Z_INDIRECT_P(retval);
				if (Z_TYPE_P(retval) == IS_UNDEF) {
					switch (type) {
						case BP_VAR_R:
							zend_error(E_NOTICE, "Undefined index: %s", ZSTR_VAL(offset_key));
						case BP_VAR_UNSET:
						case BP_VAR_IS:
							retval = &EG(uninitialized_zval);
							break;
						case BP_VAR_RW:
							zend_error(E_NOTICE,"Undefined index: %s", ZSTR_VAL(offset_key));
						case BP_VAR_W: {
							ZVAL_NULL(retval);
						}
					}
				}
			}
		} else {
			switch (type) {
				case BP_VAR_R:
					zend_error(E_NOTICE, "Undefined index: %s", ZSTR_VAL(offset_key));
				case BP_VAR_UNSET:
				case BP_VAR_IS:
					retval = &EG(uninitialized_zval);
					break;
				case BP_VAR_RW:
					zend_error(E_NOTICE,"Undefined index: %s", ZSTR_VAL(offset_key));
				case BP_VAR_W: {
					zval value;
					ZVAL_NULL(&value);
					retval = zend_symtable_update(ht, offset_key, &value);
				}
			}
		}
		return retval;
	case IS_RESOURCE:
		zend_error(E_NOTICE, "Resource ID#%d used as offset, casting to integer (%d)", Z_RES_P(offset)->handle, Z_RES_P(offset)->handle);
		index = Z_RES_P(offset)->handle;
		goto num_index;
	case IS_DOUBLE:
		index = (zend_long)Z_DVAL_P(offset);
		goto num_index;
	case IS_FALSE:
		index = 0;
		goto num_index;
	case IS_TRUE:
		index = 1;
		goto num_index;
	case IS_LONG:
		index = Z_LVAL_P(offset);
num_index:
		if ((retval = zend_hash_index_find(ht, index)) == NULL) {
			switch (type) {
				case BP_VAR_R:
					zend_error(E_NOTICE, "Undefined offset: " ZEND_LONG_FMT, index);
				case BP_VAR_UNSET:
				case BP_VAR_IS:
					retval = &EG(uninitialized_zval);
					break;
```

```
377:         case BP_VAR_RW:
378:           zend_error(E_NOTICE, "Undefined offset: " ZEND_LONG_FMT, index);
379:         case BP_VAR_W: {
380:           zval value;
381:           ZVAL_UNDEF(&value);
382:           retval = zend_hash_index_update(ht, index, &value);
383:         }
384:       }
385:     }
386:     return retval;
387:   case IS_REFERENCE:
388:     ZVAL_DEREF(offset);
389:     goto try_again;
390:   default:
391:     zend_error(E_WARNING, "Illegal offset type");
392:     return (type == BP_VAR_W || type == BP_VAR_RW) ?
393:       &EG(error_zval) : &EG(uninitialized_zval);
394:   }
395: } /* }}} */
396:
397: static int spl_array_has_dimension(zval *object, zval *offset, int check_empty);
398:
399: static zval *spl_array_read_dimension_ex(int check_inherited, zval *object, zval *offset, int type, zval *rv) /* {{{ */
400: {
401:   spl_array_object *intern = Z_SPLARRAY_P(object);
402:   zval *ret;
403:
404:   if (check_inherited &&
405:     (intern->fptr_offset_get || (type == BP_VAR_IS && intern->fptr_offset_has))) {
406:     if (type == BP_VAR_IS) {
407:       if (!spl_array_has_dimension(object, offset, 0)) {
408:         return &EG(uninitialized_zval);
409:       }
410:     }
411:
412:     if (intern->fptr_offset_get) {
413:       zval tmp;
414:       if (!offset) {
415:         ZVAL_UNDEF(&tmp);
416:         offset = &tmp;
417:       } else {
418:         SEPARATE_ARG_IF_REF(offset);
419:       }
420:       zend_call_method_with_1_params(object, Z_OBJCE_P(object), &intern->fptr_offset_get, "offsetGet", rv, offset);
421:       zval_ptr_dtor(offset);
422:
423:       if (!Z_ISUNDEF_P(rv)) {
424:         return rv;
425:       }
426:       return &EG(uninitialized_zval);
427:     }
428:   }
429:
430:   ret = spl_array_get_dimension_ptr(check_inherited, intern, offset, type);
431:
432:   /* When in a write context,
433:    * ZE has to be fooled into thinking this is in a reference set
434:    * by separating (if necessary) and returning as IS_REFERENCE (with refcount == 1)
435:    */
436:
437:   if ((type == BP_VAR_W || type == BP_VAR_RW || type == BP_VAR_UNSET) &&
438:       !Z_ISREF_P(ret) &&
439:       EXPECTED(ret != &EG(uninitialized_zval))) {
440:     ZVAL_NEW_REF(ret, ret);
441:   }
442:
443:   return ret;
444: } /* }}} */
445:
446: static zval *spl_array_read_dimension(zval *object, zval *offset, int type, zval *rv) /* {{{ */
447: {
448:   return spl_array_read_dimension_ex(1, object, offset, type, rv);
449: } /* }}} */
450:
451: static void spl_array_write_dimension_ex(int check_inherited, zval *object, zval *offset, zval *value) /* {{{ */
452: {
453:   spl_array_object *intern = Z_SPLARRAY_P(object);
454:   zend_long index;
455:   HashTable *ht;
456:
457:   if (check_inherited && intern->fptr_offset_set) {
458:     zval tmp;
459:
460:     if (!offset) {
461:       ZVAL_NULL(&tmp);
462:       offset = &tmp;
463:     } else {
464:       SEPARATE_ARG_IF_REF(offset);
465:     }
466:     zend_call_method_with_2_params(object, Z_OBJCE_P(object), &intern->fptr_offset_set, "offsetSet", NULL, offset, value);
467:     zval_ptr_dtor(offset);
468:     return;
469:   }
470:
471:   if (intern->nApplyCount > 0) {
472:     zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
473:     return;
474:   }
475:
476:   Z_TRY_ADDREF_P(value);
477:   if (!offset) {
478:     ht = spl_array_get_hash_table(intern);
479:     zend_hash_next_index_insert(ht, value);
480:     return;
481:   }
482:
483: try_again:
484:   switch (Z_TYPE_P(offset)) {
485:     case IS_STRING:
486:       ht = spl_array_get_hash_table(intern);
487:       zend_symtable_update_ind(ht, Z_STR_P(offset), value);
488:       return;
489:     case IS_DOUBLE:
490:       index = (zend_long)Z_DVAL_P(offset);
491:       goto num_index;
492:     case IS_RESOURCE:
493:       index = Z_RES_HANDLE_P(offset);
494:       goto num_index;
495:     case IS_FALSE:
496:       index = 0;
497:       goto num_index;
498:     case IS_TRUE:
499:       index = 1;
500:       goto num_index;
501:     case IS_LONG:
502:       index = Z_LVAL_P(offset);
503: num_index:
504:       ht = spl_array_get_hash_table(intern);
505:       zend_hash_index_update(ht, index, value);
506:       return;
507:     case IS_NULL:
508:       ht = spl_array_get_hash_table(intern);
509:       zend_hash_next_index_insert(ht, value);
510:       return;
511:     case IS_REFERENCE:
512:       ZVAL_DEREF(offset);
513:       goto try_again;
514:     default:
515:       zend_error(E_WARNING, "Illegal offset type");
516:       zval_ptr_dtor(value);
517:       return;
518:   }
519: } /* }}} */
520:
521: static void spl_array_write_dimension(zval *object, zval *offset, zval *value) /* {{{ */
522: {
523:   spl_array_write_dimension_ex(1, object, offset, value);
524: } /* }}} */
525:
526: static void spl_array_unset_dimension_ex(int check_inherited, zval *object, zval *offset) /* {{{ */
527: {
528:   zend_long index;
529:   HashTable *ht;
530:   spl_array_object *intern = Z_SPLARRAY_P(object);
531:
532:   if (check_inherited && intern->fptr_offset_del) {
533:     SEPARATE_ARG_IF_REF(offset);
534:     zend_call_method_with_1_params(object, Z_OBJCE_P(object), &intern->fptr_offset_del, "offsetUnset", NULL, offset);
535:     zval_ptr_dtor(offset);
536:     return;
537:   }
538:
539:   if (intern->nApplyCount > 0) {
540:     zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
541:     return;
542:   }
543:
544: try_again:
545:   switch (Z_TYPE_P(offset)) {
546:   case IS_STRING:
547:     ht = spl_array_get_hash_table(intern);
548:     if (ht == &EG(symbol_table)) {
549:       if (zend_delete_global_variable(Z_STR_P(offset))) {
550:         zend_error(E_NOTICE,"Undefined index: %s", Z_STRVAL_P(offset));
551:       }
552:     } else {
553:       zval *data = zend_symtable_find(ht, Z_STR_P(offset));
554:
555:       if (data) {
556:         if (Z_TYPE_P(data) == IS_INDIRECT) {
557:           data = Z_INDIRECT_P(data);
558:           if (Z_TYPE_P(data) == IS_UNDEF) {
559:             zend_error(E_NOTICE,"Undefined index: %s", Z_STRVAL_P(offset));
560:           } else {
561:             zval_ptr_dtor(data);
562:             ZVAL_UNDEF(data);
563:             HT_FLAGS(ht) |= HASH_FLAG_HAS_EMPTY_IND;
564:             zend_hash_move_forward_ex(ht, spl_array_get_pos_ptr(ht, intern));
565:             if (spl_array_is_object(intern)) {
566:               spl_array_skip_protected(intern, ht);
567:             }
568:           }
569:         } else if (zend_symtable_del(ht, Z_STR_P(offset)) == FAILURE) {
570:           zend_error(E_NOTICE,"Undefined index: %s", Z_STRVAL_P(offset));
571:         }
572:       } else {
573:         zend_error(E_NOTICE,"Undefined index: %s", Z_STRVAL_P(offset));
574:       }
575:     }
576:     break;
577:   case IS_DOUBLE:
578:     index = (zend_long)Z_DVAL_P(offset);
579:     goto num_index;
580:   case IS_RESOURCE:
581:     index = Z_RES_HANDLE_P(offset);
582:     goto num_index;
583:   case IS_FALSE:
584:     index = 0;
585:     goto num_index;
586:   case IS_TRUE:
587:     index = 1;
588:     goto num_index;
589:   case IS_LONG:
590:     index = Z_LVAL_P(offset);
591: num_index:
592:     ht = spl_array_get_hash_table(intern);
593:     if (zend_hash_index_del(ht, index) == FAILURE) {
594:       zend_error(E_NOTICE,"Undefined offset: " ZEND_LONG_FMT, index);
595:     }
596:     break;
597:   case IS_REFERENCE:
598:     ZVAL_DEREF(offset);
599:     goto try_again;
600:   default:
601:     zend_error(E_WARNING, "Illegal offset type");
602:     return;
603:   }
604: } /* }}} */
605:
606: static void spl_array_unset_dimension(zval *object, zval *offset) /* {{{ */
607: {
608:   spl_array_unset_dimension_ex(1, object, offset);
609: } /* }}} */
610:
611: static int spl_array_has_dimension_ex(int check_inherited, zval *object, zval *offset, int check_empty) /* {{{ */
612: {
613:   spl_array_object *intern = Z_SPLARRAY_P(object);
614:   zend_long index;
615:   zval rv, *value = NULL, *tmp;
616:
617:   if (check_inherited && intern->fptr_offset_has) {
618:     SEPARATE_ARG_IF_REF(offset);
619:     zend_call_method_with_1_params(object, Z_OBJCE_P(object), &intern->fptr_offset_has, "offsetExists", &rv, offset);
620:     zval_ptr_dtor(offset);
621:
622:     if (!Z_ISUNDEF(rv) && zend_is_true(&rv)) {
623:       zval_ptr_dtor(&rv);
624:       if (check_empty != 1) {
625:         return 1;
626:       } else if (intern->fptr_offset_get) {
627:         value = spl_array_read_dimension_ex(1, object, offset, BP_VAR_R, &rv);
628:       }
629:     } else {
630:       zval_ptr_dtor(&rv);
631:       return 0;
632:     }
633:   }
634:
635:   if (!value) {
636:     HashTable *ht = spl_array_get_hash_table(intern);
637:
638: try_again:
639:     switch (Z_TYPE_P(offset)) {
640:       case IS_STRING:
641:         if ((tmp = zend_symtable_find(ht, Z_STR_P(offset))) != NULL) {
642:           if (check_empty == 2) {
643:             return 1;
644:           }
645:         } else {
646:           return 0;
647:         }
648:         break;
649:
650:       case IS_DOUBLE:
651:         index = (zend_long)Z_DVAL_P(offset);
652:         goto num_index;
653:       case IS_RESOURCE:
654:         index = Z_RES_HANDLE_P(offset);
655:         goto num_index;
656:       case IS_FALSE:
657:         index = 0;
658:         goto num_index;
659:       case IS_TRUE:
660:         index = 1;
661:         goto num_index;
662:       case IS_LONG:
663:         index = Z_LVAL_P(offset);
664: num_index:
665:         if ((tmp = zend_hash_index_find(ht, index)) != NULL) {
666:           if (check_empty == 2) {
667:             return 1;
668:           }
669:         } else {
670:           return 0;
671:         }
672:         break;
673:       case IS_REFERENCE:
674:         ZVAL_DEREF(offset);
675:         goto try_again;
676:       default:
677:         zend_error(E_WARNING, "Illegal offset type");
678:         return 0;
679:     }
680:
681:     if (check_empty && check_inherited && intern->fptr_offset_get) {
682:       value = spl_array_read_dimension_ex(1, object, offset, BP_VAR_R, &rv);
683:     } else {
684:       value = tmp;
685:     }
686:   }
687:
688:   {
689:     zend_bool result = check_empty ? zend_is_true(value) : Z_TYPE_P(value) != IS_NULL;
690:     if (value == &rv) {
691:       zval_ptr_dtor(&rv);
692:     }
693:     return result;
694:   }
695: } /* }}} */
696:
697: static int spl_array_has_dimension(zval *object, zval *offset, int check_empty) /* {{{ */
698: {
699:   return spl_array_has_dimension_ex(1, object, offset, check_empty);
700: } /* }}} */
701:
702: /* {{{ spl_array_object_verify_pos_ex */
703: static inline int spl_array_object_verify_pos_ex(spl_array_object *object, HashTable *ht, const char *msg_prefix)
704: {
705:   if (!ht) {
706:     php_error_docref(NULL, E_NOTICE, "%sArray was modified outside object and is no longer an array", msg_prefix);
707:     return FAILURE;
708:   }
709:
710:   return SUCCESS;
711: } /* }}} */
712:
713: /* {{{ spl_array_object_verify_pos */
714: static inline int spl_array_object_verify_pos(spl_array_object *object, HashTable *ht)
715: {
716:   return spl_array_object_verify_pos_ex(object, ht, "");
717: } /* }}} */
718:
719: /* {{{ proto bool ArrayObject::offsetExists(mixed $index)
720:        proto bool ArrayIterator::offsetExists(mixed $index)
721:    Returns whether the requested $index exists. */
722: SPL_METHOD(Array, offsetExists)
723: {
724:   zval *index;
725:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &index) == FAILURE) {
726:     return;
727:   }
728:   RETURN_BOOL(spl_array_has_dimension_ex(0, getThis(), index, 2));
729: } /* }}} */
730:
731: /* {{{ proto mixed ArrayObject::offsetGet(mixed $index)
732:        proto mixed ArrayIterator::offsetGet(mixed $index)
733:    Returns the value at the specified $index. */
734: SPL_METHOD(Array, offsetGet)
735: {
736:   zval *value, *index;
737:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &index) == FAILURE) {
738:     return;
739:   }
740:   value = spl_array_read_dimension_ex(0, getThis(), index, BP_VAR_R, return_value);
741:   if (value != return_value) {
742:     ZVAL_DEREF(value);
743:     ZVAL_COPY(return_value, value);
744:   }
745: } /* }}} */
746:
747: /* {{{ proto void ArrayObject::offsetSet(mixed $index, mixed $newval)
748:        proto void ArrayIterator::offsetSet(mixed $index, mixed $newval)
749:    Sets the value at the specified $index to $newval. */
750: SPL_METHOD(Array, offsetSet)
751: {
752:   zval *index, *value;
```

```
753:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &index, &value) == FAILURE) {
754:     return;
755:   }
756:   spl_array_write_dimension_ex(0, getThis(), index, value);
757: } /* }}} */
758:
759: void spl_array_iterator_append(zval *object, zval *append_value) /* {{{ */
760: {
761:   spl_array_object *intern = Z_SPLARRAY_P(object);
762:   HashTable *aht = spl_array_get_hash_table(intern);
763:
764:   if (!aht) {
765:     php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
766:     return;
767:   }
768:
769:   if (spl_array_is_object(intern)) {
770:     zend_throw_error(NULL, "Cannot append properties to objects, use %s::offsetSet() instead", ZSTR_VAL(Z_OBJCE_P(object)->name));
771:     return;
772:   }
773:
774:   spl_array_write_dimension(object, NULL, append_value);
775: } /* }}} */
776:
777: /* {{{ proto void ArrayObject::append(mixed $newval)
778:        proto void ArrayIterator::append(mixed $newval)
779:    Appends the value (cannot be called for objects). */
780: SPL_METHOD(Array, append)
781: {
782:   zval *value;
783:
784:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &value) == FAILURE) {
785:     return;
786:   }
787:   spl_array_iterator_append(getThis(), value);
788: } /* }}} */
789:
790: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
791:        proto void ArrayIterator::offsetUnset(mixed $index)
792:    Unsets the value at the specified $index. */
793: SPL_METHOD(Array, offsetUnset)
794: {
795:   zval *index;
796:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &index) == FAILURE) {
797:     return;
798:   }
799:   spl_array_unset_dimension_ex(0, getThis(), index);
800: } /* }}} */
801:
802: /* {{{ proto array ArrayObject::getArrayCopy()
803:        proto array ArrayIterator::getArrayCopy()
804:    Return a copy of the contained array */
805: SPL_METHOD(Array, getArrayCopy)
806: {
807:   zval *object = getThis();
808:   spl_array_object *intern = Z_SPLARRAY_P(object);
809:
810:   RETURN_ARR(zend_array_dup(spl_array_get_hash_table(intern)));
811: } /* }}} */
812:
813: static HashTable *spl_array_get_properties(zval *object) /* {{{ */
814: {
815:   spl_array_object *intern = Z_SPLARRAY_P(object);
816:
817:   if (intern->ar_flags & SPL_ARRAY_STD_PROP_LIST) {
818:     if (!intern->std.properties) {
819:       rebuild_object_properties(&intern->std);
820:     }
821:     return intern->std.properties;
822:   }
823:
824:   return spl_array_get_hash_table(intern);
825: } /* }}} */
826:
827: static HashTable* spl_array_get_debug_info(zval *obj, int *is_temp) /* {{{ */
828: {
829:   zval *storage;
830:   zend_string *zname;
831:   zend_class_entry *base;
832:   spl_array_object *intern = Z_SPLARRAY_P(obj);
833:
834:   if (!intern->std.properties) {
835:     rebuild_object_properties(&intern->std);
836:   }
837:
838:   if (intern->ar_flags & SPL_ARRAY_IS_SELF) {
839:     *is_temp = 0;
840:     return intern->std.properties;
841:   } else {
842:     HashTable *debug_info;
843:     *is_temp = 1;
844:
845:     debug_info = zend_new_array(zend_hash_num_elements(intern->std.properties) + 1);
846:     zend_hash_copy(debug_info, intern->std.properties, (copy_ctor_func_t) zval_add_ref);
847:
848:     storage = &intern->array;
849:     Z_TRY_ADDREF_P(storage);
850:
851:     base = Z_OBJ_HT_P(obj) == &spl_handler_ArrayIterator
852:       ? spl_ce_ArrayIterator : spl_ce_ArrayObject;
853:     zname = spl_gen_private_prop_name(base, "storage", sizeof("storage")-1);
854:     zend_symtable_update(debug_info, zname, storage);
855:     zend_string_release(zname);
856:
857:     return debug_info;
858:   }
859: }
860: /* }}} */
861:
862: static HashTable *spl_array_get_gc(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
863: {
864:   spl_array_object *intern = Z_SPLARRAY_P(obj);
865:   *gc_data = &intern->array;
866:   *gc_data_count = 1;
867:   return zend_std_get_properties(obj);
868: }
869: /* }}} */
870:
871: static zval *spl_array_read_property(zval *object, zval *member, int type, void **cache_slot, zval *rv) /* {{{ */
872: {
873:   spl_array_object *intern = Z_SPLARRAY_P(object);
874:
875:   if ((intern->ar_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
876:     && !std_object_handlers.has_property(object, member, 2, NULL)) {
877:     return spl_array_read_dimension(object, member, type, rv);
878:   }
879:   return std_object_handlers.read_property(object, member, type, cache_slot, rv);
880: } /* }}} */
881:
882: static void spl_array_write_property(zval *object, zval *member, zval *value, void **cache_slot) /* {{{ */
883: {
884:   spl_array_object *intern = Z_SPLARRAY_P(object);
885:
886:   if ((intern->ar_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
887:     && !std_object_handlers.has_property(object, member, 2, NULL)) {
888:     spl_array_write_dimension(object, member, value);
889:     return;
890:   }
891:   std_object_handlers.write_property(object, member, value, cache_slot);
892: } /* }}} */
893:
894: static zval *spl_array_get_property_ptr_ptr(zval *object, zval *member, int type, void **cache_slot) /* {{{ */
895: {
896:   spl_array_object *intern = Z_SPLARRAY_P(object);
897:
898:   if ((intern->ar_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
899:     && !std_object_handlers.has_property(object, member, 2, NULL)) {
900:     /* If object has offsetGet() overridden, then fallback to read_property,
901:      * which will call offsetGet(). */
902:     if (intern->fptr_offset_get) {
903:       return NULL;
904:     }
905:     return spl_array_get_dimension_ptr(1, intern, member, type);
906:   }
907:   return std_object_handlers.get_property_ptr_ptr(object, member, type, cache_slot);
908: } /* }}} */
909:
910: static int spl_array_has_property(zval *object, zval *member, int has_set_exists, void **cache_slot) /* {{{ */
911: {
912:   spl_array_object *intern = Z_SPLARRAY_P(object);
913:
914:   if ((intern->ar_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
915:     && !std_object_handlers.has_property(object, member, 2, NULL)) {
916:     return spl_array_has_dimension(object, member, has_set_exists);
917:   }
918:   return std_object_handlers.has_property(object, member, has_set_exists, cache_slot);
919: } /* }}} */
920:
921: static void spl_array_unset_property(zval *object, zval *member, void **cache_slot) /* {{{ */
922: {
923:   spl_array_object *intern = Z_SPLARRAY_P(object);
924:
925:   if ((intern->ar_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0
926:     && !std_object_handlers.has_property(object, member, 2, NULL)) {
927:     spl_array_unset_dimension(object, member);
928:     return;
929:   }
930:   std_object_handlers.unset_property(object, member, cache_slot);
931: } /* }}} */
932:
933: static int spl_array_compare_objects(zval *o1, zval *o2) /* {{{ */
934: {
935:   HashTable     *ht1,
936:                 *ht2;
937:   spl_array_object  *intern1,
938:                     *intern2;
939:   int           result  = 0;
940:
941:   intern1 = Z_SPLARRAY_P(o1);
942:   intern2 = Z_SPLARRAY_P(o2);
943:   ht1   = spl_array_get_hash_table(intern1);
944:   ht2   = spl_array_get_hash_table(intern2);
945:
946:   result = zend_compare_symbol_tables(ht1, ht2);
947:   /* if we just compared std.properties, don't do it again */
948:   if (result == 0 &&
949:       !(ht1 == intern1->std.properties && ht2 == intern2->std.properties)) {
950:     result = std_object_handlers.compare_objects(o1, o2);
951:   }
952:   return result;
953: } /* }}} */
954:
955: static int spl_array_skip_protected(spl_array_object *intern, HashTable *aht) /* {{{ */
956: {
957:   zend_string *string_key;
958:   zend_ulong num_key;
959:   zval *data;
960:
961:   if (spl_array_is_object(intern)) {
962:     uint32_t *pos_ptr = spl_array_get_pos_ptr(aht, intern);
963:
964:     do {
965:       if (zend_hash_get_current_key_ex(aht, &string_key, &num_key, pos_ptr) == HASH_KEY_IS_STRING) {
966:         data = zend_hash_get_current_data_ex(aht, pos_ptr);
967:         if (data && Z_TYPE_P(data) == IS_INDIRECT &&
968:             Z_TYPE_P(data = Z_INDIRECT_P(data)) == IS_UNDEF) {
969:           /* skip */
970:         } else if (!ZSTR_LEN(string_key) || ZSTR_VAL(string_key)[0]) {
971:           return SUCCESS;
972:         }
973:       } else {
974:         return SUCCESS;
975:       }
976:       if (zend_hash_has_more_elements_ex(aht, pos_ptr) != SUCCESS) {
977:         return FAILURE;
978:       }
979:       zend_hash_move_forward_ex(aht, pos_ptr);
980:     } while (1);
981:   }
982:   return FAILURE;
983: } /* }}} */
984:
985: static int spl_array_next_ex(spl_array_object *intern, HashTable *aht) /* {{{ */
986: {
987:   uint32_t *pos_ptr = spl_array_get_pos_ptr(aht, intern);
988:
989:   zend_hash_move_forward_ex(aht, pos_ptr);
990:   if (spl_array_is_object(intern)) {
991:     return spl_array_skip_protected(intern, aht);
992:   } else {
993:     return zend_hash_has_more_elements_ex(aht, pos_ptr);
994:   }
995: } /* }}} */
996:
997: static int spl_array_next(spl_array_object *intern) /* {{{ */
998: {
999:   HashTable *aht = spl_array_get_hash_table(intern);
1000:
1001:   return spl_array_next_ex(intern, aht);
1002:
1003: } /* }}} */
1004:
1005: static void spl_array_it_dtor(zend_object_iterator *iter) /* {{{ */
1006: {
1007:   zend_user_it_invalidate_current(iter);
1008:   zval_ptr_dtor(&iter->data);
1009: }
1010: /* }}} */
1011:
1012: static int spl_array_it_valid(zend_object_iterator *iter) /* {{{ */
1013: {
1014:   spl_array_object *object = Z_SPLARRAY_P(&iter->data);
1015:   HashTable *aht = spl_array_get_hash_table(object);
1016:
1017:   if (object->ar_flags & SPL_ARRAY_OVERLOADED_VALID) {
1018:     return zend_user_it_valid(iter);
1019:   } else {
1020:     if (spl_array_object_verify_pos_ex(object, aht, "ArrayIterator::valid(): ") == FAILURE) {
1021:       return FAILURE;
1022:     }
1023:
1024:     return zend_hash_has_more_elements_ex(aht, spl_array_get_pos_ptr(aht, object));
1025:   }
1026: }
1027: /* }}} */
1028:
1029: static zval *spl_array_it_get_current_data(zend_object_iterator *iter) /* {{{ */
1030: {
1031:   spl_array_object *object = Z_SPLARRAY_P(&iter->data);
1032:   HashTable *aht = spl_array_get_hash_table(object);
1033:
1034:   if (object->ar_flags & SPL_ARRAY_OVERLOADED_CURRENT) {
1035:     return zend_user_it_get_current_data(iter);
1036:   } else {
1037:     zval *data = zend_hash_get_current_data_ex(aht, spl_array_get_pos_ptr(aht, object));
1038:     if (Z_TYPE_P(data) == IS_INDIRECT) {
1039:       data = Z_INDIRECT_P(data);
1040:     }
1041:     return data;
1042:   }
1043: }
1044: /* }}} */
1045:
1046: static void spl_array_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
1047: {
1048:   spl_array_object *object = Z_SPLARRAY_P(&iter->data);
1049:   HashTable *aht = spl_array_get_hash_table(object);
1050:
1051:   if (object->ar_flags & SPL_ARRAY_OVERLOADED_KEY) {
1052:     zend_user_it_get_current_key(iter, key);
1053:   } else {
1054:     if (spl_array_object_verify_pos_ex(object, aht, "ArrayIterator::current(): ") == FAILURE) {
1055:       ZVAL_NULL(key);
1056:     } else {
1057:       zend_hash_get_current_key_zval_ex(aht, key, spl_array_get_pos_ptr(aht, object));
1058:     }
1059:   }
1060: }
1061: /* }}} */
1062:
1063: static void spl_array_it_move_forward(zend_object_iterator *iter) /* {{{ */
1064: {
1065:   spl_array_object *object = Z_SPLARRAY_P(&iter->data);
1066:   HashTable *aht = spl_array_get_hash_table(object);
1067:
1068:   if (object->ar_flags & SPL_ARRAY_OVERLOADED_NEXT) {
1069:     zend_user_it_move_forward(iter);
1070:   } else {
1071:     zend_user_it_invalidate_current(iter);
1072:     if (!aht) {
1073:       php_error_docref(NULL, E_NOTICE, "ArrayIterator::current(): Array was modified outside object and is no longer an array");
1074:       return;
1075:     }
1076:
1077:     spl_array_next_ex(object, aht);
1078:   }
1079: }
1080: /* }}} */
1081:
1082: static void spl_array_rewind(spl_array_object *intern) /* {{{ */
1083: {
1084:   HashTable *aht = spl_array_get_hash_table(intern);
1085:
1086:   if (!aht) {
1087:     php_error_docref(NULL, E_NOTICE, "ArrayIterator::rewind(): Array was modified outside object and is no longer an array");
1088:     return;
1089:   }
1090:
1091:   if (intern->ht_iter == (uint32_t)-1) {
1092:     spl_array_get_pos_ptr(aht, intern);
1093:   } else {
1094:     zend_hash_internal_pointer_reset_ex(aht, spl_array_get_pos_ptr(aht, intern));
1095:     spl_array_skip_protected(intern, aht);
1096:   }
1097: }
1098: /* }}} */
1099:
1100: static void spl_array_it_rewind(zend_object_iterator *iter) /* {{{ */
1101: {
1102:   spl_array_object *object = Z_SPLARRAY_P(&iter->data);
1103:
1104:   if (object->ar_flags & SPL_ARRAY_OVERLOADED_REWIND) {
1105:     zend_user_it_rewind(iter);
1106:   } else {
1107:     zend_user_it_invalidate_current(iter);
1108:     spl_array_rewind(object);
1109:   }
1110: }
1111: /* }}} */
1112:
1113: /* {{{ spl_array_set_array */
1114: static void spl_array_set_array(zval *object, spl_array_object *intern, zval *array, zend_long ar_flags, int just_array) {
1115:   if (Z_TYPE_P(array) != IS_OBJECT && Z_TYPE_P(array) != IS_ARRAY) {
1116:     zend_throw_exception(spl_ce_InvalidArgumentException, "Passed variable is not an array or object", 0);
1117:     return;
1118:   }
1119:
1120:   if (Z_TYPE_P(array) == IS_ARRAY) {
1121:     zval_ptr_dtor(&intern->array);
1122:     if (Z_REFCOUNT_P(array) == 1) {
1123:       ZVAL_COPY(&intern->array, array);
1124:     } else {
1125:       //??? TODO: try to avoid array duplication
1126:       ZVAL_ARR(&intern->array, zend_array_dup(Z_ARR_P(array)));
1127:     }
1128:   } else {
```

```
1129:     if (Z_OBJ_HT_P(array) == &spl_handler_ArrayObject || Z_OBJ_HT_P(array) == &spl_handler_ArrayIterator) {
1130:       zval_ptr_dtor(&intern->array);
1131:       if (just_array) {
1132:         spl_array_object *other = Z_SPLARRAY_P(array);
1133:         ar_flags = other->ar_flags & ~SPL_ARRAY_INT_MASK;
1134:       }
1135:       if (Z_OBJ_P(object) == Z_OBJ_P(array)) {
1136:         ar_flags |= SPL_ARRAY_IS_SELF;
1137:         ZVAL_UNDEF(&intern->array);
1138:       } else {
1139:         ar_flags |= SPL_ARRAY_USE_OTHER;
1140:         ZVAL_COPY(&intern->array, array);
1141:       }
1142:     } else {
1143:       zend_object_get_properties_t handler = Z_OBJ_HANDLER_P(array, get_properties);
1144:       if (handler != std_object_handlers.get_properties) {
1145:         zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0,
1146:           "Overloaded object of type %s is not compatible with %s",
1147:           ZSTR_VAL(Z_OBJCE_P(array)->name), ZSTR_VAL(intern->std.ce->name));
1148:         return;
1149:       }
1150:       zval_ptr_dtor(&intern->array);
1151:       ZVAL_COPY(&intern->array, array);
1152:     }
1153:   }
1154:
1155:   intern->ar_flags &= ~SPL_ARRAY_IS_SELF & ~SPL_ARRAY_USE_OTHER;
1156:   intern->ar_flags |= ar_flags;
1157:   intern->ht_iter = (uint32_t)-1;
1158: }
1159: /* }}} */
1160:
1161: /* iterator handler table */
1162: static const zend_object_iterator_funcs spl_array_it_funcs = {
1163:   spl_array_it_dtor,
1164:   spl_array_it_valid,
1165:   spl_array_it_get_current_data,
1166:   spl_array_it_get_current_key,
1167:   spl_array_it_move_forward,
1168:   spl_array_it_rewind,
1169:   NULL
1170: };
1171:
1172: zend_object_iterator *spl_array_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1173: {
1174:   zend_user_iterator *iterator;
1175:   spl_array_object *array_object = Z_SPLARRAY_P(object);
1176:
1177:   if (by_ref && (array_object->ar_flags & SPL_ARRAY_OVERLOADED_CURRENT)) {
1178:     zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1179:     return NULL;
1180:   }
1181:
1182:   iterator = emalloc(sizeof(zend_user_iterator));
1183:
1184:   zend_iterator_init(&iterator->it);
1185:
1186:   ZVAL_COPY(&iterator->it.data, object);
1187:   iterator->it.funcs = &spl_array_it_funcs;
1188:   iterator->ce = ce;
1189:   ZVAL_UNDEF(&iterator->value);
1190:
1191:   return &iterator->it;
1192: }
1193: /* }}} */
1194:
1195: /* {{{ proto void ArrayObject::__construct([array|object ar = array() [, int flags = 0 [, string iterator_class = "ArrayIterator"]]])
1196:    Constructs a new array object from an array or object. */
1197: SPL_METHOD(Array, __construct)
1198: {
1199:   zval *object = getThis();
1200:   spl_array_object *intern;
1201:   zval *array;
1202:   zend_long ar_flags = 0;
1203:   zend_class_entry *ce_get_iterator = spl_ce_Iterator;
1204:
1205:   if (ZEND_NUM_ARGS() == 0) {
1206:     return; /* nothing to do */
1207:   }
1208:
1209:   if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "z|lC", &array, &ar_flags, &ce_get_iterator) == FAILURE) {
1210:     return;
1211:   }
1212:
1213:   intern = Z_SPLARRAY_P(object);
1214:
1215:   if (ZEND_NUM_ARGS() > 2) {
1216:     intern->ce_get_iterator = ce_get_iterator;
1217:   }
1218:
1219:   ar_flags &= ~SPL_ARRAY_INT_MASK;
1220:
1221:   spl_array_set_array(object, intern, array, ar_flags, ZEND_NUM_ARGS() == 1);
1222: }
1223:  /* }}} */
1224:
1225: /* {{{ proto void ArrayIterator::__construct([array|object ar = array() [, int flags = 0]])
1226:    Constructs a new array iterator from an array or object. */
1227: SPL_METHOD(ArrayIterator, __construct)
1228: {
1229:   zval *object = getThis();
1230:   spl_array_object *intern;
1231:   zval *array;
1232:   zend_long ar_flags = 0;
1233:
1234:   if (ZEND_NUM_ARGS() == 0) {
1235:     return; /* nothing to do */
1236:   }
1237:
1238:   if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "z|l", &array, &ar_flags) == FAILURE) {
1239:     return;
1240:   }
1241:
1242:   intern = Z_SPLARRAY_P(object);
1243:
1244:   ar_flags &= ~SPL_ARRAY_INT_MASK;
1245:
1246:   spl_array_set_array(object, intern, array, ar_flags, ZEND_NUM_ARGS() == 1);
1247: }
1248:  /* }}} */
1249:
1250: /* {{{ proto void ArrayObject::setIteratorClass(string iterator_class)
1251:    Set the class used in getIterator. */
1252: SPL_METHOD(Array, setIteratorClass)
1253: {
1254:   zval *object = getThis();
1255:   spl_array_object *intern = Z_SPLARRAY_P(object);
1256:   zend_class_entry * ce_get_iterator = spl_ce_Iterator;
1257:
1258:   ZEND_PARSE_PARAMETERS_START(1, 1)
1259:     Z_PARAM_CLASS(ce_get_iterator)
1260:   ZEND_PARSE_PARAMETERS_END();
1261:
1262:   intern->ce_get_iterator = ce_get_iterator;
1263: }
1264: /* }}} */
1265:
1266: /* {{{ proto string ArrayObject::getIteratorClass()
1267:    Get the class used in getIterator. */
1268: SPL_METHOD(Array, getIteratorClass)
1269: {
1270:   zval *object = getThis();
1271:   spl_array_object *intern = Z_SPLARRAY_P(object);
1272:
1273:   if (zend_parse_parameters_none() == FAILURE) {
1274:     return;
1275:   }
1276:
1277:   zend_string_addref(intern->ce_get_iterator->name);
1278:   RETURN_STR(intern->ce_get_iterator->name);
1279: }
1280: /* }}} */
1281:
1282: /* {{{ proto int ArrayObject::getFlags()
1283:    Get flags */
1284: SPL_METHOD(Array, getFlags)
1285: {
1286:   zval *object = getThis();
1287:   spl_array_object *intern = Z_SPLARRAY_P(object);
1288:
1289:   if (zend_parse_parameters_none() == FAILURE) {
1290:     return;
1291:   }
1292:
1293:   RETURN_LONG(intern->ar_flags & ~SPL_ARRAY_INT_MASK);
1294: }
1295: /* }}} */
1296:
1297: /* {{{ proto void ArrayObject::setFlags(int flags)
1298:    Set flags */
1299: SPL_METHOD(Array, setFlags)
1300: {
1301:   zval *object = getThis();
1302:   spl_array_object *intern = Z_SPLARRAY_P(object);
1303:   zend_long ar_flags = 0;
1304:
1305:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &ar_flags) == FAILURE) {
1306:     return;
1307:   }
1308:
1309:   intern->ar_flags = (intern->ar_flags & SPL_ARRAY_INT_MASK) | (ar_flags & ~SPL_ARRAY_INT_MASK);
1310: }
1311: /* }}} */
1312:
1313: /* {{{ proto Array|Object ArrayObject::exchangeArray(Array|Object ar = array())
1314:    Replace the referenced array or object with a new one and return the old one (right now copy - to be changed) */
1315: SPL_METHOD(Array, exchangeArray)
1316: {
1317:   zval *object = getThis(), *array;
1318:   spl_array_object *intern = Z_SPLARRAY_P(object);
1319:
1320:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &array) == FAILURE) {
1321:     return;
1322:   }
1323:
1324:   if (intern->nApplyCount > 0) {
1325:     zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
1326:     return;
1327:   }
1328:
1329:   RETVAL_ARR(zend_array_dup(spl_array_get_hash_table(intern)));
1330:   spl_array_set_array(object, intern, array, 0L, 1);
1331: }
1332: /* }}} */
1333:
1334: /* {{{ proto ArrayIterator ArrayObject::getIterator()
1335:    Create a new iterator from a ArrayObject instance */
1336: SPL_METHOD(Array, getIterator)
1337: {
1338:   zval *object = getThis();
1339:   spl_array_object *intern = Z_SPLARRAY_P(object);
1340:   HashTable *aht = spl_array_get_hash_table(intern);
1341:
1342:   if (zend_parse_parameters_none() == FAILURE) {
1343:     return;
1344:   }
1345:
1346:   if (!aht) {
1347:     php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
1348:     return;
1349:   }
1350:
1351:   ZVAL_OBJ(return_value, spl_array_object_new_ex(intern->ce_get_iterator, object, 0));
1352: }
1353: /* }}} */
1354:
1355: /* {{{ proto void ArrayIterator::rewind()
1356:    Rewind array back to the start */
1357: SPL_METHOD(Array, rewind)
1358: {
1359:   zval *object = getThis();
1360:   spl_array_object *intern = Z_SPLARRAY_P(object);
1361:
1362:   if (zend_parse_parameters_none() == FAILURE) {
1363:     return;
1364:   }
1365:
1366:   spl_array_rewind(intern);
1367: }
1368: /* }}} */
1369:
1370: /* {{{ proto void ArrayIterator::seek(int $position)
1371:    Seek to position. */
1372: SPL_METHOD(Array, seek)
1373: {
1374:   zend_long opos, position;
1375:   zval *object = getThis();
1376:   spl_array_object *intern = Z_SPLARRAY_P(object);
1377:   HashTable *aht = spl_array_get_hash_table(intern);
1378:   int result;
1379:
1380:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &position) == FAILURE) {
1381:     return;
1382:   }
1383:
1384:   if (!aht) {
1385:     php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
1386:     return;
1387:   }
1388:
1389:   opos = position;
1390:
1391:   if (position >= 0) { /* negative values are not supported */
1392:     spl_array_rewind(intern);
1393:     result = SUCCESS;
1394:
1395:     while (position-- > 0 && (result = spl_array_next(intern)) == SUCCESS);
1396:
1397:     if (result == SUCCESS && zend_hash_has_more_elements_ex(aht, spl_array_get_pos_ptr(aht, intern)) == SUCCESS) {
1398:       return; /* ok */
1399:     }
1400:   }
1401:   zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Seek position " ZEND_LONG_FMT " is out of range", opos);
1402: } /* }}} */
1403:
1404: static int spl_array_object_count_elements_helper(spl_array_object *intern, zend_long *count) /* {{{ */
1405: {
1406:   HashTable *aht = spl_array_get_hash_table(intern);
1407:   HashPosition pos, *pos_ptr;
1408:
1409:   if (!aht) {
1410:     php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
1411:     *count = 0;
1412:     return FAILURE;
1413:   }
1414:
1415:   if (spl_array_is_object(intern)) {
1416:     /* We need to store the 'pos' since we'll modify it in the functions
1417:      * we're going to call and which do not support 'pos' as parameter. */
1418:     pos_ptr = spl_array_get_pos_ptr(aht, intern);
1419:     pos = *pos_ptr;
1420:     *count = 0;
1421:     spl_array_rewind(intern);
1422:     while (*pos_ptr != HT_INVALID_IDX && spl_array_next(intern) == SUCCESS) {
1423:       (*count)++;
1424:     }
1425:     *pos_ptr = pos;
1426:     return SUCCESS;
1427:   } else {
1428:     *count = zend_hash_num_elements(aht);
1429:     return SUCCESS;
1430:   }
1431: } /* }}} */
1432:
1433: int spl_array_object_count_elements(zval *object, zend_long *count) /* {{{ */
1434: {
1435:   spl_array_object *intern = Z_SPLARRAY_P(object);
1436:
1437:   if (intern->fptr_count) {
1438:     zval rv;
1439:     zend_call_method_with_0_params(object, intern->std.ce, &intern->fptr_count, "count", &rv);
1440:     if (Z_TYPE(rv) != IS_UNDEF) {
1441:       *count = zval_get_long(&rv);
1442:       zval_ptr_dtor(&rv);
1443:       return SUCCESS;
1444:     }
1445:     *count = 0;
1446:     return FAILURE;
1447:   }
1448:   return spl_array_object_count_elements_helper(intern, count);
1449: } /* }}} */
1450:
1451: /* {{{ proto int ArrayObject::count()
1452:        proto int ArrayIterator::count()
1453:    Return the number of elements in the Iterator. */
1454: SPL_METHOD(Array, count)
1455: {
1456:   zend_long count;
1457:   spl_array_object *intern = Z_SPLARRAY_P(getThis());
1458:
1459:   if (zend_parse_parameters_none() == FAILURE) {
1460:     return;
1461:   }
1462:
1463:   spl_array_object_count_elements_helper(intern, &count);
1464:
1465:   RETURN_LONG(count);
1466: } /* }}} */
1467:
1468: static void spl_array_method(INTERNAL_FUNCTION_PARAMETERS, char *fname, int fname_len, int use_arg) /* {{{ */
1469: {
1470:   spl_array_object *intern = Z_SPLARRAY_P(getThis());
1471:   HashTable *aht = spl_array_get_hash_table(intern);
1472:   zval function_name, params[2], *arg = NULL;
1473:
1474:   ZVAL_STRINGL(&function_name, fname, fname_len);
1475:
1476:   ZVAL_NEW_EMPTY_REF(&params[0]);
1477:   ZVAL_ARR(Z_REFVAL(params[0]), aht);
1478:   GC_ADDREF(aht);
1479:
1480:   if (!use_arg) {
1481:     intern->nApplyCount++;
1482:     call_user_function_ex(EG(function_table), NULL, &function_name, return_value, 1, params, 1, NULL);
1483:     intern->nApplyCount--;
1484:   } else if (use_arg == SPL_ARRAY_METHOD_MAY_USER_ARG) {
1485:     if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "|z", &arg) == FAILURE) {
1486:       zend_throw_exception(spl_ce_BadMethodCallException, "Function expects one argument at most", 0);
1487:       goto exit;
1488:     }
1489:     if (arg) {
1490:       ZVAL_COPY_VALUE(&params[1], arg);
1491:     }
1492:     intern->nApplyCount++;
1493:     call_user_function_ex(EG(function_table), NULL, &function_name, return_value, arg ? 2 : 1, params, 1, NULL);
1494:     intern->nApplyCount--;
1495:   } else {
1496:     if (ZEND_NUM_ARGS() != 1 || zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "z", &arg) == FAILURE) {
1497:       zend_throw_exception(spl_ce_BadMethodCallException, "Function expects exactly one argument", 0);
1498:       goto exit;
1499:     }
1500:     ZVAL_COPY_VALUE(&params[1], arg);
1501:     intern->nApplyCount++;
1502:     call_user_function_ex(EG(function_table), NULL, &function_name, return_value, 2, params, 1, NULL);
1503:     intern->nApplyCount--;
1504:   }
```

```
1505:
1506: exit:
1507:   {
1508:     HashTable *new_ht = Z_ARRVAL_P(Z_REFVAL(params[0]));
1509:     if (aht != new_ht) {
1510:       spl_array_replace_hash_table(intern, new_ht);
1511:     } else {
1512:       GC_DELREF(aht);
1513:     }
1514:     efree(Z_REF(params[0]));
1515:     zend_string_free(Z_STR(function_name));
1516:   }
1517: } /* }}} */
1518:
1519: #define SPL_ARRAY_METHOD(cname, fname, use_arg) \
1520: SPL_METHOD(cname, fname) \
1521: { \
1522:   spl_array_method(INTERNAL_FUNCTION_PARAM_PASSTHRU, #fname, sizeof(#fname)-1, use_arg); \
1523: }
1524:
1525: /* {{{ proto int ArrayObject::asort([int $sort_flags = SORT_REGULAR ])
1526:        proto int ArrayIterator::asort([int $sort_flags = SORT_REGULAR ])
1527:   Sort the entries by values. */
1528: SPL_ARRAY_METHOD(Array, asort, SPL_ARRAY_METHOD_MAY_USER_ARG) /* }}} */
1529:
1530: /* {{{ proto int ArrayObject::ksort([int $sort_flags = SORT_REGULAR ])
1531:        proto int ArrayIterator::ksort([int $sort_flags = SORT_REGULAR ])
1532:   Sort the entries by key. */
1533: SPL_ARRAY_METHOD(Array, ksort, SPL_ARRAY_METHOD_MAY_USER_ARG) /* }}} */
1534:
1535: /* {{{ proto int ArrayObject::uasort(callback cmp_function)
1536:        proto int ArrayIterator::uasort(callback cmp_function)
1537:   Sort the entries by values user defined function. */
1538: SPL_ARRAY_METHOD(Array, uasort, SPL_ARRAY_METHOD_USE_ARG) /* }}} */
1539:
1540: /* {{{ proto int ArrayObject::uksort(callback cmp_function)
1541:        proto int ArrayIterator::uksort(callback cmp_function)
1542:   Sort the entries by key using user defined function. */
1543: SPL_ARRAY_METHOD(Array, uksort, SPL_ARRAY_METHOD_USE_ARG) /* }}} */
1544:
1545: /* {{{ proto int ArrayObject::natsort()
1546:        proto int ArrayIterator::natsort()
1547:   Sort the entries by values using "natural order" algorithm. */
1548: SPL_ARRAY_METHOD(Array, natsort, SPL_ARRAY_METHOD_NO_ARG) /* }}} */
1549:
1550: /* {{{ proto int ArrayObject::natcasesort()
1551:        proto int ArrayIterator::natcasesort()
1552:   Sort the entries by key using case insensitive "natural order" algorithm. */
1553: SPL_ARRAY_METHOD(Array, natcasesort, SPL_ARRAY_METHOD_NO_ARG) /* }}} */
1554:
1555: /* {{{ proto mixed|NULL ArrayIterator::current()
1556:   Return current array entry */
1557: SPL_METHOD(Array, current)
1558: {
1559:   zval *object = getThis();
1560:   spl_array_object *intern = Z_SPLARRAY_P(object);
1561:   zval *entry;
1562:   HashTable *aht = spl_array_get_hash_table(intern);
1563:
1564:   if (zend_parse_parameters_none() == FAILURE) {
1565:     return;
1566:   }
1567:
1568:   if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1569:     return;
1570:   }
1571:
1572:   if ((entry = zend_hash_get_current_data_ex(aht, spl_array_get_pos_ptr(aht, intern))) == NULL) {
1573:     return;
1574:   }
1575:   if (Z_TYPE_P(entry) == IS_INDIRECT) {
1576:     entry = Z_INDIRECT_P(entry);
1577:     if (Z_TYPE_P(entry) == IS_UNDEF) {
1578:       return;
1579:     }
1580:   }
1581:   ZVAL_DEREF(entry);
1582:   ZVAL_COPY(return_value, entry);
1583: }
1584: /* }}} */
1585:
1586: /* {{{ proto mixed|NULL ArrayIterator::key()
1587:   Return current array key */
1588: SPL_METHOD(Array, key)
1589: {
1590:   if (zend_parse_parameters_none() == FAILURE) {
1591:     return;
1592:   }
1593:
1594:   spl_array_iterator_key(getThis(), return_value);
1595: } /* }}} */
1596:
1597: void spl_array_iterator_key(zval *object, zval *return_value) /* {{{ */
1598: {
1599:   spl_array_object *intern = Z_SPLARRAY_P(object);
1600:   HashTable *aht = spl_array_get_hash_table(intern);
1601:
1602:   if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1603:     return;
1604:   }
1605:
1606:   zend_hash_get_current_key_zval_ex(aht, return_value, spl_array_get_pos_ptr(aht, intern));
1607: }
1608: /* }}} */
1609:
1610: /* {{{ proto void ArrayIterator::next()
1611:   Move to next entry */
1612: SPL_METHOD(Array, next)
1613: {
1614:   zval *object = getThis();
1615:   spl_array_object *intern = Z_SPLARRAY_P(object);
1616:   HashTable *aht = spl_array_get_hash_table(intern);
1617:
1618:   if (zend_parse_parameters_none() == FAILURE) {
1619:     return;
1620:   }
1621:
1622:   if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1623:     return;
1624:   }
1625:
1626:   spl_array_next_ex(intern, aht);
1627: }
1628: /* }}} */
1629:
1630: /* {{{ proto bool ArrayIterator::valid()
1631:   Check whether array contains more entries */
1632: SPL_METHOD(Array, valid)
1633: {
1634:   zval *object = getThis();
1635:   spl_array_object *intern = Z_SPLARRAY_P(object);
1636:   HashTable *aht = spl_array_get_hash_table(intern);
1637:
1638:   if (zend_parse_parameters_none() == FAILURE) {
1639:     return;
1640:   }
1641:
1642:   if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1643:     RETURN_FALSE;
1644:   } else {
1645:     RETURN_BOOL(zend_hash_has_more_elements_ex(aht, spl_array_get_pos_ptr(aht, intern)) == SUCCESS);
1646:   }
1647: }
1648: /* }}} */
1649:
1650: /* {{{ proto bool RecursiveArrayIterator::hasChildren()
1651:   Check whether current element has children (e.g. is an array) */
1652: SPL_METHOD(Array, hasChildren)
1653: {
1654:   zval *object = getThis(), *entry;
1655:   spl_array_object *intern = Z_SPLARRAY_P(object);
1656:   HashTable *aht = spl_array_get_hash_table(intern);
1657:
1658:   if (zend_parse_parameters_none() == FAILURE) {
1659:     return;
1660:   }
1661:
1662:   if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1663:     RETURN_FALSE;
1664:   }
1665:
1666:   if ((entry = zend_hash_get_current_data_ex(aht, spl_array_get_pos_ptr(aht, intern))) == NULL) {
1667:     RETURN_FALSE;
1668:   }
1669:
1670:   if (Z_TYPE_P(entry) == IS_INDIRECT) {
1671:     entry = Z_INDIRECT_P(entry);
1672:   }
1673:
1674:   ZVAL_DEREF(entry);
1675:   RETURN_BOOL(Z_TYPE_P(entry) == IS_ARRAY || (Z_TYPE_P(entry) == IS_OBJECT && (intern->ar_flags & SPL_ARRAY_CHILD_ARRAYS_ONLY) == 0));
1676: }
1677: /* }}} */
1678:
1679: /* {{{ proto object RecursiveArrayIterator::getChildren()
1680:   Create a sub iterator for the current element (same class as $this) */
1681: SPL_METHOD(Array, getChildren)
1682: {
1683:   zval *object = getThis(), *entry, flags;
1684:   spl_array_object *intern = Z_SPLARRAY_P(object);
1685:   HashTable *aht = spl_array_get_hash_table(intern);
1686:
1687:   if (zend_parse_parameters_none() == FAILURE) {
1688:     return;
1689:   }
1690:
1691:   if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1692:     return;
1693:   }
1694:
1695:   if ((entry = zend_hash_get_current_data_ex(aht, spl_array_get_pos_ptr(aht, intern))) == NULL) {
1696:     return;
1697:   }
1698:
1699:   if (Z_TYPE_P(entry) == IS_INDIRECT) {
1700:     entry = Z_INDIRECT_P(entry);
1701:   }
1702:
1703:   ZVAL_DEREF(entry);
1704:   if (Z_TYPE_P(entry) == IS_OBJECT) {
1705:     if ((intern->ar_flags & SPL_ARRAY_CHILD_ARRAYS_ONLY) != 0) {
1706:       return;
1707:     }
1708:     if (instanceof_function(Z_OBJCE_P(entry), Z_OBJCE_P(getThis()))) {
1709:       ZVAL_OBJ(return_value, Z_OBJ_P(entry));
1710:       Z_ADDREF_P(return_value);
1711:       return;
1712:     }
1713:   }
1714:
1715:   ZVAL_LONG(&flags, intern->ar_flags);
1716:   spl_instantiate_arg_ex2(Z_OBJCE_P(getThis()), return_value, entry, &flags);
1717: }
1718: /* }}} */
1719:
1720: /* {{{ proto string ArrayObject::serialize()
1721:   Serialize the object */
1722: SPL_METHOD(Array, serialize)
1723: {
1724:   zval *object = getThis();
1725:   spl_array_object *intern = Z_SPLARRAY_P(object);
1726:   HashTable *aht = spl_array_get_hash_table(intern);
1727:   zval members, flags;
1728:   php_serialize_data_t var_hash;
1729:   smart_str buf = {0};
1730:
1731:   if (zend_parse_parameters_none() == FAILURE) {
1732:     return;
1733:   }
1734:
1735:   if (!aht) {
1736:     php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
1737:     return;
1738:   }
1739:
1740:   PHP_VAR_SERIALIZE_INIT(var_hash);
1741:
1742:   ZVAL_LONG(&flags, (intern->ar_flags & SPL_ARRAY_CLONE_MASK));
1743:
1744:   /* storage */
1745:   smart_str_appendl(&buf, "x:", 2);
1746:   php_var_serialize(&buf, &flags, &var_hash);
1747:
1748:   if (!(intern->ar_flags & SPL_ARRAY_IS_SELF)) {
1749:     php_var_serialize(&buf, &intern->array, &var_hash);
1750:     smart_str_appendc(&buf, ';');
1751:   }
1752:
1753:   /* members */
1754:   smart_str_appendl(&buf, "m:", 2);
1755:   if (!intern->std.properties) {
1756:     rebuild_object_properties(&intern->std);
1757:   }
1758:
1759:   ZVAL_ARR(&members, intern->std.properties);
1760:
1761:   php_var_serialize(&buf, &members, &var_hash); /* finishes the string */
1762:
1763:   /* done */
1764:   PHP_VAR_SERIALIZE_DESTROY(var_hash);
1765:
1766:   if (buf.s) {
1767:     RETURN_NEW_STR(buf.s);
1768:   }
1769:
1770:   RETURN_NULL();
1771: } /* }}} */
1772:
1773: /* {{{ proto void ArrayObject::unserialize(string serialized)
1774:  * unserialize the object
1775:  */
1776: SPL_METHOD(Array, unserialize)
1777: {
1778:   zval *object = getThis();
1779:   spl_array_object *intern = Z_SPLARRAY_P(object);
1780:
1781:   char *buf;
1782:   size_t buf_len;
1783:   const unsigned char *p, *s;
1784:   php_unserialize_data_t var_hash;
1785:   zval *members, *pflags, *array;
1786:   zend_long flags;
1787:
1788:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &buf, &buf_len) == FAILURE) {
1789:     return;
1790:   }
1791:
1792:   if (buf_len == 0) {
1793:     return;
1794:   }
1795:
1796:   if (intern->nApplyCount > 0) {
1797:     zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
1798:     return;
1799:   }
1800:
1801:   /* storage */
1802:   s = p = (const unsigned char*)buf;
1803:   PHP_VAR_UNSERIALIZE_INIT(var_hash);
1804:
1805:   if (*p!= 'x' || *++p != ':') {
1806:     goto outexcept;
1807:   }
1808:   ++p;
1809:
1810:   zflags = var_tmp_var(&var_hash);
1811:   if (!php_var_unserialize(zflags, &p, s + buf_len, &var_hash) || Z_TYPE_P(zflags) != IS_LONG) {
1812:     goto outexcept;
1813:   }
1814:
1815:   --p; /* for ';' */
1816:   flags = Z_LVAL_P(zflags);
1817:   /* flags needs to be verified and we also need to verify whether the next
1818:    * thing we get is ';'. After that we require an 'm' or something else
1819:    * where 'm' stands for members and anything else should be an array. If
1820:    * neither 'a' or 'm' follows we have an error. */
1821:
1822:   if (*p != ';') {
1823:     goto outexcept;
1824:   }
1825:   ++p;
1826:
1827:   if (flags & SPL_ARRAY_IS_SELF) {
1828:     /* If IS_SELF is used, the flags are not followed by an array/object */
1829:     intern->ar_flags &= ~SPL_ARRAY_CLONE_MASK;
1830:     intern->ar_flags |= flags & SPL_ARRAY_CLONE_MASK;
1831:     zval_ptr_dtor(&intern->array);
1832:     ZVAL_UNDEF(&intern->array);
1833:   } else {
1834:     if (*p!='a' && *p!='O' && *p!='C' && *p!='r') {
1835:       goto outexcept;
1836:     }
1837:
1838:     array = var_tmp_var(&var_hash);
1839:     if (!php_var_unserialize(array, &p, s + buf_len, &var_hash)
1840:         || (Z_TYPE_P(array) != IS_ARRAY && Z_TYPE_P(array) != IS_OBJECT)) {
1841:       goto outexcept;
1842:     }
1843:
1844:     intern->ar_flags &= ~SPL_ARRAY_CLONE_MASK;
1845:     intern->ar_flags |= flags & SPL_ARRAY_CLONE_MASK;
1846:
1847:     if (Z_TYPE_P(array) == IS_ARRAY) {
1848:       zval_ptr_dtor(&intern->array);
1849:       ZVAL_COPY(&intern->array, array);
1850:     } else {
1851:       spl_array_set_array(object, intern, array, 0L, 1);
1852:     }
1853:
1854:     if (*p != ';') {
1855:       goto outexcept;
1856:     }
1857:       ++p;
1858:   }
1859:
1860:   /* members */
1861:   if (*p!= 'm' || *++p != ':') {
1862:     goto outexcept;
1863:   }
1864:   ++p;
1865:
1866:   members = var_tmp_var(&var_hash);
1867:   if (!php_var_unserialize(members, &p, s + buf_len, &var_hash) || Z_TYPE_P(members) != IS_ARRAY) {
1868:     goto outexcept;
1869:   }
1870:
1871:   /* copy members */
1872:   object_properties_load(&intern->std, Z_ARRVAL_P(members));
1873:
1874:   /* done reading $serialized */
1875:   PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
1876:   return;
1877:
1878: outexcept:
1879:   PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
1880:   zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Error at offset " ZEND_LONG_FMT " of %zd bytes", (zend_long)((char*)p - buf), buf_len);
```

```
1881:   return;
1882:
1883: } /* }}} */
1884:
1885: /* {{{ arginfo and function table */
1886: ZEND_BEGIN_ARG_INFO_EX(arginfo_array___construct, 0, 0, 0)
1887:   ZEND_ARG_INFO(0, array)
1888:   ZEND_ARG_INFO(0, ar_flags)
1889:   ZEND_ARG_INFO(0, iterator_class)
1890: ZEND_END_ARG_INFO()
1891:
1892: /* ArrayIterator::__construct and ArrayObject::__construct have different signatures */
1893: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_iterator___construct, 0, 0, 0)
1894:   ZEND_ARG_INFO(0, array)
1895:   ZEND_ARG_INFO(0, ar_flags)
1896: ZEND_END_ARG_INFO()
1897:
1898: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_offsetGet, 0, 0, 1)
1899:   ZEND_ARG_INFO(0, index)
1900: ZEND_END_ARG_INFO()
1901:
1902: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_offsetSet, 0, 0, 2)
1903:   ZEND_ARG_INFO(0, index)
1904:   ZEND_ARG_INFO(0, newval)
1905: ZEND_END_ARG_INFO()
1906:
1907: ZEND_BEGIN_ARG_INFO(arginfo_array_append, 0)
1908:   ZEND_ARG_INFO(0, value)
1909: ZEND_END_ARG_INFO()
1910:
1911: ZEND_BEGIN_ARG_INFO(arginfo_array_seek, 0)
1912:   ZEND_ARG_INFO(0, position)
1913: ZEND_END_ARG_INFO()
1914:
1915: ZEND_BEGIN_ARG_INFO(arginfo_array_exchangeArray, 0)
1916:   ZEND_ARG_INFO(0, array)
1917: ZEND_END_ARG_INFO()
1918:
1919: ZEND_BEGIN_ARG_INFO(arginfo_array_setFlags, 0)
1920:   ZEND_ARG_INFO(0, flags)
1921: ZEND_END_ARG_INFO()
1922:
1923: ZEND_BEGIN_ARG_INFO(arginfo_array_setIteratorClass, 0)
1924:   ZEND_ARG_INFO(0, iteratorClass)
1925: ZEND_END_ARG_INFO()
1926:
1927: ZEND_BEGIN_ARG_INFO(arginfo_array_uXsort, 0)
1928:   ZEND_ARG_INFO(0, cmp_function)
1929: ZEND_END_ARG_INFO();
1930:
1931: ZEND_BEGIN_ARG_INFO(arginfo_array_unserialize, 0)
1932:   ZEND_ARG_INFO(0, serialized)
1933: ZEND_END_ARG_INFO();
1934:
1935: ZEND_BEGIN_ARG_INFO(arginfo_array_void, 0)
1936: ZEND_END_ARG_INFO()
1937:
1938: static const zend_function_entry spl_funcs_ArrayObject[] = {
1939:   SPL_ME(Array, __construct,      arginfo_array___construct,      ZEND_ACC_PUBLIC)
1940:   SPL_ME(Array, offsetExists,     arginfo_array_offsetGet,        ZEND_ACC_PUBLIC)
1941:   SPL_ME(Array, offsetGet,        arginfo_array_offsetGet,        ZEND_ACC_PUBLIC)
1942:   SPL_ME(Array, offsetSet,        arginfo_array_offsetSet,        ZEND_ACC_PUBLIC)
1943:   SPL_ME(Array, offsetUnset,      arginfo_array_offsetGet,        ZEND_ACC_PUBLIC)
1944:   SPL_ME(Array, append,           arginfo_array_append,           ZEND_ACC_PUBLIC)
1945:   SPL_ME(Array, getArrayCopy,     arginfo_array_void,             ZEND_ACC_PUBLIC)
1946:   SPL_ME(Array, count,            arginfo_array_void,             ZEND_ACC_PUBLIC)
1947:   SPL_ME(Array, getFlags,         arginfo_array_void,             ZEND_ACC_PUBLIC)
1948:   SPL_ME(Array, setFlags,         arginfo_array_setFlags,         ZEND_ACC_PUBLIC)
1949:   SPL_ME(Array, asort,            arginfo_array_void,             ZEND_ACC_PUBLIC)
1950:   SPL_ME(Array, ksort,            arginfo_array_void,             ZEND_ACC_PUBLIC)
1951:   SPL_ME(Array, uasort,           arginfo_array_uXsort,           ZEND_ACC_PUBLIC)
1952:   SPL_ME(Array, uksort,           arginfo_array_uXsort,           ZEND_ACC_PUBLIC)
1953:   SPL_ME(Array, natsort,          arginfo_array_void,             ZEND_ACC_PUBLIC)
1954:   SPL_ME(Array, natcasesort,      arginfo_array_void,             ZEND_ACC_PUBLIC)
1955:   SPL_ME(Array, unserialize,      arginfo_array_unserialize,      ZEND_ACC_PUBLIC)
1956:   SPL_ME(Array, serialize,        arginfo_array_void,             ZEND_ACC_PUBLIC)
1957:   /* ArrayObject specific */
1958:   SPL_ME(Array, getIterator,      arginfo_array_void,             ZEND_ACC_PUBLIC)
1959:   SPL_ME(Array, exchangeArray,    arginfo_array_exchangeArray,    ZEND_ACC_PUBLIC)
1960:   SPL_ME(Array, setIteratorClass, arginfo_array_setIteratorClass, ZEND_ACC_PUBLIC)
1961:   SPL_ME(Array, getIteratorClass, arginfo_array_void,             ZEND_ACC_PUBLIC)
1962:   PHP_FE_END
1963: };
1964:
1965: static const zend_function_entry spl_funcs_ArrayIterator[] = {
1966:   SPL_ME(ArrayIterator, __construct, arginfo_array_iterator___construct,     ZEND_ACC_PUBLIC)
1967:   SPL_ME(Array, offsetExists,     arginfo_array_offsetGet,        ZEND_ACC_PUBLIC)
1968:   SPL_ME(Array, offsetGet,        arginfo_array_offsetGet,        ZEND_ACC_PUBLIC)
1969:   SPL_ME(Array, offsetSet,        arginfo_array_offsetSet,        ZEND_ACC_PUBLIC)
1970:   SPL_ME(Array, offsetUnset,      arginfo_array_offsetGet,        ZEND_ACC_PUBLIC)
1971:   SPL_ME(Array, append,           arginfo_array_append,           ZEND_ACC_PUBLIC)
1972:   SPL_ME(Array, getArrayCopy,     arginfo_array_void,             ZEND_ACC_PUBLIC)
1973:   SPL_ME(Array, count,            arginfo_array_void,             ZEND_ACC_PUBLIC)
1974:   SPL_ME(Array, getFlags,         arginfo_array_void,             ZEND_ACC_PUBLIC)
1975:   SPL_ME(Array, setFlags,         arginfo_array_setFlags,         ZEND_ACC_PUBLIC)
1976:   SPL_ME(Array, asort,            arginfo_array_void,             ZEND_ACC_PUBLIC)
1977:   SPL_ME(Array, ksort,            arginfo_array_void,             ZEND_ACC_PUBLIC)
1978:   SPL_ME(Array, uasort,           arginfo_array_uXsort,           ZEND_ACC_PUBLIC)
1979:   SPL_ME(Array, uksort,           arginfo_array_uXsort,           ZEND_ACC_PUBLIC)
1980:   SPL_ME(Array, natsort,          arginfo_array_void,             ZEND_ACC_PUBLIC)
1981:   SPL_ME(Array, natcasesort,      arginfo_array_void,             ZEND_ACC_PUBLIC)
1982:   SPL_ME(Array, unserialize,      arginfo_array_unserialize,      ZEND_ACC_PUBLIC)
1983:   SPL_ME(Array, serialize,        arginfo_array_void,             ZEND_ACC_PUBLIC)
1984:   /* ArrayIterator specific */
1985:   SPL_ME(Array, rewind,           arginfo_array_void,             ZEND_ACC_PUBLIC)
1986:   SPL_ME(Array, current,          arginfo_array_void,             ZEND_ACC_PUBLIC)
1987:   SPL_ME(Array, key,              arginfo_array_void,             ZEND_ACC_PUBLIC)
1988:   SPL_ME(Array, next,             arginfo_array_void,             ZEND_ACC_PUBLIC)
1989:   SPL_ME(Array, valid,            arginfo_array_void,             ZEND_ACC_PUBLIC)
1990:   SPL_ME(Array, seek,             arginfo_array_seek,             ZEND_ACC_PUBLIC)
1991:   PHP_FE_END
1992: };
1993:
1994: static const zend_function_entry spl_funcs_RecursiveArrayIterator[] = {
1995:   SPL_ME(Array, hasChildren,   arginfo_array_void, ZEND_ACC_PUBLIC)
1996:   SPL_ME(Array, getChildren,   arginfo_array_void, ZEND_ACC_PUBLIC)
1997:   PHP_FE_END
1998: };
1999: /* }}} */
2000:
2001: /* {{{ PHP_MINIT_FUNCTION(spl_array) */
2002: PHP_MINIT_FUNCTION(spl_array)
2003: {
2004:   REGISTER_SPL_STD_CLASS_EX(ArrayObject, spl_array_object_new, spl_funcs_ArrayObject);
2005:   REGISTER_SPL_IMPLEMENTS(ArrayObject, Aggregate);
2006:   REGISTER_SPL_IMPLEMENTS(ArrayObject, ArrayAccess);
2007:   REGISTER_SPL_IMPLEMENTS(ArrayObject, Serializable);
2008:   REGISTER_SPL_IMPLEMENTS(ArrayObject, Countable);
2009:   memcpy(&spl_handler_ArrayObject, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
2010:
2011:   spl_handler_ArrayObject.offset = XtOffsetOf(spl_array_object, std);
2012:
2013:   spl_handler_ArrayObject.clone_obj = spl_array_object_clone;
2014:   spl_handler_ArrayObject.read_dimension = spl_array_read_dimension;
2015:   spl_handler_ArrayObject.write_dimension = spl_array_write_dimension;
2016:   spl_handler_ArrayObject.unset_dimension = spl_array_unset_dimension;
2017:   spl_handler_ArrayObject.has_dimension = spl_array_has_dimension;
2018:   spl_handler_ArrayObject.count_elements = spl_array_object_count_elements;
2019:
2020:   spl_handler_ArrayObject.get_properties = spl_array_get_properties;
2021:   spl_handler_ArrayObject.get_debug_info = spl_array_get_debug_info;
2022:   spl_handler_ArrayObject.get_gc = spl_array_get_gc;
2023:   spl_handler_ArrayObject.read_property = spl_array_read_property;
2024:   spl_handler_ArrayObject.write_property = spl_array_write_property;
2025:   spl_handler_ArrayObject.get_property_ptr_ptr = spl_array_get_property_ptr_ptr;
2026:   spl_handler_ArrayObject.has_property = spl_array_has_property;
2027:   spl_handler_ArrayObject.unset_property = spl_array_unset_property;
2028:
2029:   spl_handler_ArrayObject.compare_objects = spl_array_compare_objects;
2030:   spl_handler_ArrayObject.dtor_obj = zend_objects_destroy_object;
2031:   spl_handler_ArrayObject.free_obj = spl_array_object_free_storage;
2032:
2033:   REGISTER_SPL_STD_CLASS_EX(ArrayIterator, spl_array_object_new, spl_funcs_ArrayIterator);
2034:   REGISTER_SPL_IMPLEMENTS(ArrayIterator, Iterator);
2035:   REGISTER_SPL_IMPLEMENTS(ArrayIterator, ArrayAccess);
2036:   REGISTER_SPL_IMPLEMENTS(ArrayIterator, SeekableIterator);
2037:   REGISTER_SPL_IMPLEMENTS(ArrayIterator, Serializable);
2038:   REGISTER_SPL_IMPLEMENTS(ArrayIterator, Countable);
2039:   memcpy(&spl_handler_ArrayIterator, &spl_handler_ArrayObject, sizeof(zend_object_handlers));
2040:   spl_ce_ArrayIterator->get_iterator = spl_array_get_iterator;
2041:
2042:   REGISTER_SPL_CLASS_CONST_LONG(ArrayObject,   "STD_PROP_LIST",   SPL_ARRAY_STD_PROP_LIST);
2043:   REGISTER_SPL_CLASS_CONST_LONG(ArrayObject,   "ARRAY_AS_PROPS",  SPL_ARRAY_ARRAY_AS_PROPS);
2044:
2045:   REGISTER_SPL_CLASS_CONST_LONG(ArrayIterator, "STD_PROP_LIST",   SPL_ARRAY_STD_PROP_LIST);
2046:   REGISTER_SPL_CLASS_CONST_LONG(ArrayIterator, "ARRAY_AS_PROPS",  SPL_ARRAY_ARRAY_AS_PROPS);
2047:
2048:   REGISTER_SPL_SUB_CLASS_EX(RecursiveArrayIterator, ArrayIterator, spl_array_object_new, spl_funcs_RecursiveArrayIterator);
2049:   REGISTER_SPL_IMPLEMENTS(RecursiveArrayIterator, RecursiveIterator);
2050:   spl_ce_RecursiveArrayIterator->get_iterator = spl_array_get_iterator;
2051:
2052:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveArrayIterator, "CHILD_ARRAYS_ONLY", SPL_ARRAY_CHILD_ARRAYS_ONLY);
2053:
2054:   return SUCCESS;
2055: }
2056: /* }}} */
2057:
2058: /*
2059:  * Local variables:
2060:  * tab-width: 4
2061:  * c-basic-offset: 4
2062:  * End:
2063:  * vim600: fdm=marker
2064:  * vim: noet sw=4 ts=4
2065:  */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Etienne Kneuss <colder@php.net>                             |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_HEAP_H
#define SPL_HEAP_H

#include "php.h"
#include "php_spl.h"

extern PHPAPI zend_class_entry *spl_ce_SplHeap;
extern PHPAPI zend_class_entry *spl_ce_SplMinHeap;
extern PHPAPI zend_class_entry *spl_ce_SplMaxHeap;

extern PHPAPI zend_class_entry *spl_ce_SplPriorityQueue;

PHP_MINIT_FUNCTION(spl_heap);

#endif /* SPL_HEAP_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_ENGINE_H
#define SPL_ENGINE_H

#include "php.h"
#include "php_spl.h"
#include "zend_interfaces.h"

PHPAPI void spl_instantiate(zend_class_entry *pce, zval *object);

PHPAPI zend_long spl_offset_convert_to_long(zval *offset);

/* {{{ spl_instantiate_arg_ex1 */
static inline int spl_instantiate_arg_ex1(zend_class_entry *pce, zval *retval, zval *arg1)
{
   zend_function *func = pce->constructor;
   spl_instantiate(pce, retval);

   zend_call_method(retval, pce, &func, ZSTR_VAL(func->common.function_name), ZSTR_LEN(func->common.function_name), NULL, 1, arg1, NULL);
   return 0;
}
/* }}} */

/* {{{ spl_instantiate_arg_ex2 */
static inline int spl_instantiate_arg_ex2(zend_class_entry *pce, zval *retval, zval *arg1, zval *arg2)
{
   zend_function *func = pce->constructor;
   spl_instantiate(pce, retval);

   zend_call_method(retval, pce, &func, ZSTR_VAL(func->common.function_name), ZSTR_LEN(func->common.function_name), NULL, 2, arg1, arg2);
   return 0;
}
/* }}} */

/* {{{ spl_instantiate_arg_n */
static inline void spl_instantiate_arg_n(zend_class_entry *pce, zval *retval, int argc, zval *argv)
{
   zend_function *func = pce->constructor;
   zend_fcall_info fci;
   zend_fcall_info_cache fcc;
   zval dummy;

   spl_instantiate(pce, retval);

   fci.size = sizeof(zend_fcall_info);
   ZVAL_STR(&fci.function_name, func->common.function_name);
   fci.object = Z_OBJ_P(retval);
   fci.retval = &dummy;
   fci.param_count = argc;
   fci.params = argv;
   fci.no_separation = 1;

   fcc.function_handler = func;
   fcc.calling_scope = zend_get_executed_scope();
   fcc.called_scope = pce;
   fcc.object = Z_OBJ_P(retval);

   zend_call_function(&fci, &fcc);
}
/* }}} */

#endif /* SPL_ENGINE_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "zend_exceptions.h"
#include "zend_interfaces.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_iterators.h"
#include "spl_directory.h"
#include "spl_array.h"
#include "spl_exceptions.h"
#include "zend_smart_str.h"

#ifdef accept
#undef accept
#endif

PHPAPI zend_class_entry *spl_ce_RecursiveIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveIteratorIterator;
PHPAPI zend_class_entry *spl_ce_FilterIterator;
PHPAPI zend_class_entry *spl_ce_CallbackFilterIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveFilterIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveCallbackFilterIterator;
PHPAPI zend_class_entry *spl_ce_ParentIterator;
PHPAPI zend_class_entry *spl_ce_SeekableIterator;
PHPAPI zend_class_entry *spl_ce_LimitIterator;
PHPAPI zend_class_entry *spl_ce_CachingIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveCachingIterator;
PHPAPI zend_class_entry *spl_ce_OuterIterator;
PHPAPI zend_class_entry *spl_ce_IteratorIterator;
PHPAPI zend_class_entry *spl_ce_NoRewindIterator;
PHPAPI zend_class_entry *spl_ce_InfiniteIterator;
PHPAPI zend_class_entry *spl_ce_EmptyIterator;
PHPAPI zend_class_entry *spl_ce_AppendIterator;
PHPAPI zend_class_entry *spl_ce_RegexIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveRegexIterator;
PHPAPI zend_class_entry *spl_ce_RecursiveTreeIterator;

ZEND_BEGIN_ARG_INFO(arginfo_recursive_it_void, 0)
ZEND_END_ARG_INFO()

static const zend_function_entry spl_funcs_RecursiveIterator[] = {
    SPL_ABSTRACT_ME(RecursiveIterator, hasChildren,  arginfo_recursive_it_void)
    SPL_ABSTRACT_ME(RecursiveIterator, getChildren,  arginfo_recursive_it_void)
    PHP_FE_END
};

typedef enum {
    RIT_LEAVES_ONLY = 0,
    RIT_SELF_FIRST  = 1,
    RIT_CHILD_FIRST = 2
} RecursiveIteratorMode;

#define RIT_CATCH_GET_CHILD CIT_CATCH_GET_CHILD

typedef enum {
    RTIT_BYPASS_CURRENT = 4,
    RTIT_BYPASS_KEY     = 8
} RecursiveTreeIteratorFlags;

typedef enum {
    RS_NEXT  = 0,
    RS_TEST  = 1,
    RS_SELF  = 2,
    RS_CHILD = 3,
    RS_START = 4
} RecursiveIteratorState;

typedef struct _spl_sub_iterator {
    zend_object_iterator    *iterator;
    zval                    zobject;
    zend_class_entry        *ce;
    RecursiveIteratorState  state;
} spl_sub_iterator;

typedef struct _spl_recursive_it_object {
    spl_sub_iterator         *iterators;
    int                      level;
    RecursiveIteratorMode    mode;
    int                      flags;
    int                      max_depth;
    zend_bool                in_iteration;
    zend_function            *beginIteration;
    zend_function            *endIteration;
    zend_function            *callHasChildren;
    zend_function            *callGetChildren;
    zend_function            *beginChildren;
    zend_function            *endChildren;
    zend_function            *nextElement;
    zend_class_entry         *ce;
    smart_str                prefix[6];
    smart_str                postfix[1];
    zend_object              std;
} spl_recursive_it_object;

typedef struct _spl_recursive_it_iterator {
    zend_object_iterator   intern;
} spl_recursive_it_iterator;

static zend_object_handlers spl_handlers_rec_it_it;
static zend_object_handlers spl_handlers_dual_it;

static inline spl_recursive_it_object *spl_recursive_it_from_obj(zend_object *obj) /* {{{ */ {
    return (spl_recursive_it_object*)((char*)(obj) - XtOffsetOf(spl_recursive_it_object, std));
}
/* }}} */

#define Z_SPLRECURSIVE_IT_P(zv)  spl_recursive_it_from_obj(Z_OBJ_P((zv)))

#define SPL_FETCH_AND_CHECK_DUAL_IT(var, objzval)                                \
    do {                                                                         \
        spl_dual_it_object *it = Z_SPLDUAL_IT_P(objzval);                        \
        if (it->dit_type == DIT_Unknown) {                                       \
            zend_throw_exception_ex(spl_ce_LogicException, 0,                    \
                "The object is in an invalid state as the parent constructor was not called");  \
            return;                                                              \
        }                                                                        \
        (var) = it;                                                              \
    } while (0)

#define SPL_FETCH_SUB_ELEMENT(var, object, element) \
    do { \
        if(!(object)->iterators) { \
            zend_throw_exception_ex(spl_ce_LogicException, 0, \
                "The object is in an invalid state as the parent constructor was not called"); \
            return; \
        } \
        (var) = (object)->iterators[(object)->level].element; \
    } while (0)

#define SPL_FETCH_SUB_ELEMENT_ADDR(var, object, element) \
    do { \
        if(!(object)->iterators) { \
            zend_throw_exception_ex(spl_ce_LogicException, 0, \
                "The object is in an invalid state as the parent constructor was not called"); \
            return; \
        } \
        (var) = &(object)->iterators[(object)->level].element; \
    } while (0)

#define SPL_FETCH_SUB_ITERATOR(var, object) SPL_FETCH_SUB_ELEMENT(var, object, iterator)


static void spl_recursive_it_dtor(zend_object_iterator *_iter)
{
    spl_recursive_it_iterator *iter   = (spl_recursive_it_iterator*)_iter;
    spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(iter->intern.data);
    zend_object_iterator      *sub_iter;

    while (object->level > 0) {
        if (!Z_ISUNDEF(object->iterators[object->level].zobject)) {
            sub_iter = object->iterators[object->level].iterator;
            zend_iterator_dtor(sub_iter);
            zval_ptr_dtor(&object->iterators[object->level].zobject);
        }
        object->level--;
    }
    object->iterators = erealloc(object->iterators, sizeof(spl_sub_iterator));
    object->level = 0;

    zval_ptr_dtor(&iter->intern.data);
}


static int spl_recursive_it_valid_ex(spl_recursive_it_object *object, zval *zthis)
{
    zend_object_iterator      *sub_iter;
    int                       level = object->level;

    if(!object->iterators) {
        return FAILURE;
    }
    while (level >=0) {
        sub_iter = object->iterators[level].iterator;
        if (sub_iter->funcs->valid(sub_iter) == SUCCESS) {
            return SUCCESS;
        }
        level--;
    }
    if (object->endIteration && object->in_iteration) {
        zend_call_method_with_0_params(zthis, object->ce, &object->endIteration, "endIteration", NULL);
    }
    object->in_iteration = 0;
    return FAILURE;
}

static int spl_recursive_it_valid(zend_object_iterator *iter)
{
    return spl_recursive_it_valid_ex(Z_SPLRECURSIVE_IT_P(&iter->data), &iter->data);
}

static zval *spl_recursive_it_get_current_data(zend_object_iterator *iter)
{
    spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(&iter->data);
    zend_object_iterator *sub_iter = object->iterators[object->level].iterator;

    return sub_iter->funcs->get_current_data(sub_iter);
}

static void spl_recursive_it_get_current_key(zend_object_iterator *iter, zval *key)
{
    spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(&iter->data);
    zend_object_iterator *sub_iter = object->iterators[object->level].iterator;

    if (sub_iter->funcs->get_current_key) {
        sub_iter->funcs->get_current_key(sub_iter, key);
    } else {
        ZVAL_LONG(key, iter->index);
    }
}

static void spl_recursive_it_move_forward_ex(spl_recursive_it_object *object, zval *zthis)
{
    zend_object_iterator      *iterator;
    zval                      *zobject;
    zend_class_entry          *ce;
    zval                      retval, child;
    zend_object_iterator      *sub_iter;
    int                       has_children;

    SPL_FETCH_SUB_ITERATOR(iterator, object);

    while (!EG(exception)) {
next_step:
        iterator = object->iterators[object->level].iterator;
        switch (object->iterators[object->level].state) {
            case RS_NEXT:
                iterator->funcs->move_forward(iterator);
                if (EG(exception)) {
                    if (!(object->flags & RIT_CATCH_GET_CHILD)) {
                        return;
                    } else {
                        zend_clear_exception();
                    }
                }
                /* fall through */
            case RS_START:
                if (iterator->funcs->valid(iterator) == FAILURE) {
                    break;
                }
                object->iterators[object->level].state = RS_TEST;
                /* break; */
            case RS_TEST:
                ce = object->iterators[object->level].ce;
                zobject = &object->iterators[object->level].zobject;
                if (object->callHasChildren) {
                    zend_call_method_with_0_params(zthis, object->ce, &object->callHasChildren, "callHasChildren", &retval);
                } else {
                    zend_call_method_with_0_params(zobject, ce, NULL, "haschildren", &retval);
                }
                if (EG(exception)) {
                    if (!(object->flags & RIT_CATCH_GET_CHILD)) {
                        object->iterators[object->level].state = RS_NEXT;
                        return;
                    } else {
                        zend_clear_exception();
                    }
                }
                if (Z_TYPE(retval) != IS_UNDEF) {
                    has_children = zend_is_true(&retval);
                    zval_ptr_dtor(&retval);
                    if (has_children) {
                        if (object->max_depth == -1 || object->max_depth > object->level) {
                            switch (object->mode) {
                                case RIT_LEAVES_ONLY:
                                case RIT_CHILD_FIRST:
                                    object->iterators[object->level].state = RS_CHILD;
                                    goto next_step;
                                case RIT_SELF_FIRST:
                                    object->iterators[object->level].state = RS_SELF;
                                    goto next_step;
                            }
                        } else {
                            /* do not recurse into */
                            if (object->mode == RIT_LEAVES_ONLY) {
                                /* this is not a leave, so skip it */
                                object->iterators[object->level].state = RS_NEXT;
                                goto next_step;
                            }
                        }
                    }
                }
                if (object->nextElement) {
                    zend_call_method_with_0_params(zthis, object->ce, &object->nextElement, "nextelement", NULL);
                }
                object->iterators[object->level].state = RS_NEXT;
                if (EG(exception)) {
                    if (!(object->flags & RIT_CATCH_GET_CHILD)) {
                        return;
                    } else {
                        zend_clear_exception();
                    }
                }
                return /* self */;
            case RS_SELF:
                if (object->nextElement && (object->mode == RIT_SELF_FIRST || object->mode == RIT_CHILD_FIRST)) {
                    zend_call_method_with_0_params(zthis, object->ce, &object->nextElement, "nextelement", NULL);
                }
                if (object->mode == RIT_SELF_FIRST) {
                    object->iterators[object->level].state = RS_CHILD;
                } else {
                    object->iterators[object->level].state = RS_NEXT;
                }
                return /* self */;
            case RS_CHILD:
                ce = object->iterators[object->level].ce;
                zobject = &object->iterators[object->level].zobject;
                if (object->callGetChildren) {
                    zend_call_method_with_0_params(zthis, object->ce, &object->callGetChildren, "callGetChildren", &child);
                } else {
                    zend_call_method_with_0_params(zobject, ce, NULL, "getchildren", &child);
                }

                if (EG(exception)) {
                    if (!(object->flags & RIT_CATCH_GET_CHILD)) {
                        return;
                    } else {
                        zend_clear_exception();
                        zval_ptr_dtor(&child);
                        object->iterators[object->level].state = RS_NEXT;
                        goto next_step;
                    }
                }

                if (Z_TYPE(child) == IS_UNDEF || Z_TYPE(child) != IS_OBJECT ||
                    !(ce = Z_OBJCE(child)) && instanceof_function(ce, spl_ce_RecursiveIterator)) {
                    zval_ptr_dtor(&child);
                    zend_throw_exception(spl_ce_UnexpectedValueException, "Objects returned by RecursiveIterator::getChildren() must implement RecursiveIterator", 0);
                    return;
                }

                if (object->mode == RIT_CHILD_FIRST) {
                    object->iterators[object->level].state = RS_SELF;
                } else {
                    object->iterators[object->level].state = RS_NEXT;
                }
                object->iterators = erealloc(object->iterators, sizeof(spl_sub_iterator) * (++object->level+1));
                sub_iter = ce->get_iterator(ce, &child, 0);
                ZVAL_COPY_VALUE(&object->iterators[object->level].zobject, &child);
                object->iterators[object->level].iterator = sub_iter;
                object->iterators[object->level].ce = ce;
                object->iterators[object->level].state = RS_START;
                if (sub_iter->funcs->rewind) {
                    sub_iter->funcs->rewind(sub_iter);
                }
                if (object->beginChildren) {
                    zend_call_method_with_0_params(zthis, object->ce, &object->beginChildren, "beginchildren", NULL);
                    if (EG(exception)) {
                        if (!(object->flags & RIT_CATCH_GET_CHILD)) {
                            return;
```

```
376:             } else {
377:               zend_clear_exception();
378:             }
379:           }
380:         }
381:         goto next_step;
382:       }
383:       /* no more elements */
384:       if (object->level > 0) {
385:         if (object->endChildren) {
386:           zend_call_method_with_0_params(zthis, object->ce, &object->endChildren, "endchildren", NULL);
387:           if (EG(exception)) {
388:             if (!(object->flags & RIT_CATCH_GET_CHILD)) {
389:               return;
390:             } else {
391:               zend_clear_exception();
392:             }
393:           }
394:         }
395:         if (object->level > 0) {
396:           zval garbage;
397:           ZVAL_COPY_VALUE(&garbage, &object->iterators[object->level].zobject);
398:           ZVAL_UNDEF(&object->iterators[object->level].zobject);
399:           zval_ptr_dtor(&garbage);
400:           zend_iterator_dtor(tIterator);
401:           object->level--;
402:         }
403:       } else {
404:         return; /* done completeley */
405:       }
406:     }
407: }
408:
409: static void spl_recursive_it_rewind_ex(spl_recursive_it_object *object, zval *zthis)
410: {
411:   zend_object_iterator *sub_iter;
412:
413:   SPL_FETCH_SUB_ITERATOR(sub_iter, object);
414:
415:   while (object->level) {
416:     sub_iter = object->iterators[object->level].iterator;
417:     zend_iterator_dtor(sub_iter);
418:     zval_ptr_dtor(&object->iterators[object->level--].zobject);
419:     if (!EG(exception) && (!object->endChildren || object->endChildren->common.scope != spl_ce_RecursiveIteratorIterator)) {
420:       zend_call_method_with_0_params(zthis, object->ce, &object->endChildren, "endchildren", NULL);
421:     }
422:   }
423:   object->iterators = erealloc(object->iterators, sizeof(spl_sub_iterator));
424:   object->iterators[0].state = RS_START;
425:   sub_iter = object->iterators[0].iterator;
426:   if (sub_iter->funcs->rewind) {
427:     sub_iter->funcs->rewind(sub_iter);
428:   }
429:   if (!EG(exception) && object->beginIteration && !object->in_iteration) {
430:     zend_call_method_with_0_params(zthis, object->ce, &object->beginIteration, "beginIteration", NULL);
431:   }
432:   object->in_iteration = 1;
433:   spl_recursive_it_move_forward_ex(object, zthis);
434: }
435:
436: static void spl_recursive_it_move_forward(zend_object_iterator *iter)
437: {
438:   spl_recursive_it_move_forward_ex(Z_SPLRECURSIVE_IT_P(&iter->data), &iter->data);
439: }
440:
441: static void spl_recursive_it_rewind(zend_object_iterator *iter)
442: {
443:   spl_recursive_it_rewind_ex(Z_SPLRECURSIVE_IT_P(&iter->data), &iter->data);
444: }
445:
446: static zend_object_iterator *spl_recursive_it_get_iterator(zend_class_entry *ce, zval *zobject, int by_ref)
447: {
448:   spl_recursive_it_iterator *iterator;
449:   spl_recursive_it_object *object;
450:
451:   if (by_ref) {
452:     zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
453:     return NULL;
454:   }
455:   iterator = emalloc(sizeof(spl_recursive_it_iterator));
456:   object   = Z_SPLRECURSIVE_IT_P(zobject);
457:   if (object->iterators == NULL) {
458:     zend_error(E_ERROR, "The object to be iterated is in an invalid state: "
459:         "the parent constructor has not been called");
460:   }
461:
462:   zend_iterator_init((zend_object_iterator*)iterator);
463:
464:   ZVAL_COPY(&iterator->intern.data, zobject);
465:   iterator->intern.funcs = ce->iterator_funcs.funcs;
466:   return (zend_object_iterator*)iterator;
467: }
468:
469: static const zend_object_iterator_funcs spl_recursive_it_iterator_funcs = {
470:   spl_recursive_it_dtor,
471:   spl_recursive_it_valid,
472:   spl_recursive_it_get_current_data,
473:   spl_recursive_it_get_current_key,
474:   spl_recursive_it_move_forward,
475:   spl_recursive_it_rewind,
476:   NULL
477: };
478:
479: static void spl_recursive_it_it_construct(INTERNAL_FUNCTION_PARAMETERS, zend_class_entry *ce_base, zend_class_entry *ce_inner, recursive_it_it_type rit
_type)
480: {
481:   zval *object = getThis();
482:   spl_recursive_it_object *intern;
483:   zval *iterator;
484:   zend_class_entry *ce_iterator;
485:   zend_long mode, flags;
486:   zend_error_handling error_handling;
487:   zval caching_it, aggregate_retval;
488:
489:   zend_replace_error_handling(EH_THROW, spl_ce_InvalidArgumentException, &error_handling);
490:
491:   switch (rit_type) {
492:     case RIT_RecursiveTreeIterator: {
493:       zval caching_it_flags, *user_caching_it_flags = NULL;
494:       mode = RIT_SELF_FIRST;
495:       flags = RTIT_BYPASS_KEY;
496:
497:       if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "o|lz", &iterator, &flags, &user_caching_it_flags, &mode) == SUCCESS) {
498:         if (instanceof_function(Z_OBJCE_P(iterator), zend_ce_aggregate)) {
499:           zend_call_method_with_0_params(iterator, Z_OBJCE_P(iterator), &Z_OBJCE_P(iterator)->iterator_funcs.zf_new_iterator, "getiterator", &aggregate
_retval);
500:           iterator = &aggregate_retval;
501:         } else {
502:           Z_ADDREF_P(iterator);
503:         }
504:
505:         if (user_caching_it_flags) {
506:           ZVAL_COPY(&caching_it_flags, user_caching_it_flags);
507:         } else {
508:           ZVAL_LONG(&caching_it_flags, CIT_CATCH_GET_CHILD);
509:         }
510:         spl_instantiate_arg_ex2(spl_ce_RecursiveCachingIterator, &caching_it, iterator, &caching_it_flags);
511:         zval_ptr_dtor(&caching_it_flags);
512:
513:         zval_ptr_dtor(iterator);
514:         iterator = &caching_it;
515:       } else {
516:         iterator = NULL;
517:       }
518:       break;
519:     }
520:     case RIT_RecursiveIteratorIterator:
521:     default: {
522:       mode = RIT_LEAVES_ONLY;
523:       flags = 0;
524:
525:       if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "o|ll", &iterator, &mode, &flags) == SUCCESS) {
526:         if (instanceof_function(Z_OBJCE_P(iterator), zend_ce_aggregate)) {
527:           zend_call_method_with_0_params(iterator, Z_OBJCE_P(iterator), &Z_OBJCE_P(iterator)->iterator_funcs.zf_new_iterator, "getiterator", &aggregate
_retval);
528:           iterator = &aggregate_retval;
529:         } else {
530:           Z_ADDREF_P(iterator);
531:         }
532:       } else {
533:         iterator = NULL;
534:       }
535:       break;
536:     }
537:   }
538:   if (!iterator || !instanceof_function(Z_OBJCE_P(iterator), spl_ce_RecursiveIterator)) {
539:     if (iterator) {
540:       zval_ptr_dtor(iterator);
541:     }
542:     zend_throw_exception(spl_ce_InvalidArgumentException, "An instance of RecursiveIterator or IteratorAggregate creating it is required", 0);
543:     zend_restore_error_handling(&error_handling);
544:     return;
545:   }
546:
547:   intern = Z_SPLRECURSIVE_IT_P(object);
548:   intern->iterators = emalloc(sizeof(spl_sub_iterator));
549:   intern->level = 0;
550:   intern->mode = mode;
551:   intern->flags = (int)flags;
552:   intern->max_depth = -1;
553:   intern->in_iteration = 0;
554:   intern->ce = Z_OBJCE_P(object);
555:
556:   intern->beginIteration = zend_hash_str_find_ptr(&intern->ce->function_table, "beginiteration", sizeof("beginiteration") - 1);
557:   if (intern->beginIteration->common.scope == ce_base) {
558:     intern->beginIteration = NULL;
559:   }
560:   intern->endIteration = zend_hash_str_find_ptr(&intern->ce->function_table, "enditeration", sizeof("enditeration") - 1);
561:   if (intern->endIteration->common.scope == ce_base) {
562:     intern->endIteration = NULL;
563:   }
564:   intern->callHasChildren = zend_hash_str_find_ptr(&intern->ce->function_table, "callhaschildren", sizeof("callHasChildren") - 1);
565:   if (intern->callHasChildren->common.scope == ce_base) {
566:     intern->callHasChildren = NULL;
567:   }
568:   intern->callGetChildren = zend_hash_str_find_ptr(&intern->ce->function_table, "callgetchildren", sizeof("callGetChildren") - 1);
569:   if (intern->callGetChildren->common.scope == ce_base) {
570:     intern->callGetChildren = NULL;
571:   }
572:   intern->beginChildren = zend_hash_str_find_ptr(&intern->ce->function_table, "beginchildren", sizeof("beginchildren") - 1);
573:   if (intern->beginChildren->common.scope == ce_base) {
574:     intern->beginChildren = NULL;
575:   }
576:   intern->endChildren = zend_hash_str_find_ptr(&intern->ce->function_table, "endchildren", sizeof("endchildren") - 1);
577:   if (intern->endChildren->common.scope == ce_base) {
578:     intern->endChildren = NULL;
579:   }
580:   intern->nextElement = zend_hash_str_find_ptr(&intern->ce->function_table, "nextelement", sizeof("nextElement") - 1);
581:   if (intern->nextElement->common.scope == ce_base) {
582:     intern->nextElement = NULL;
583:   }
584:
585:   ce_iterator = Z_OBJCE_P(iterator); /* respect inheritance, don't use spl_ce_RecursiveIterator */
586:   intern->iterators[0].iterator = ce_iterator->get_iterator(ce_iterator, iterator, 0);
587:   ZVAL_COPY_VALUE(&intern->iterators[0].zobject, iterator);
588:   intern->iterators[0].ce = ce_iterator;
589:   intern->iterators[0].state = RS_START;
590:
591:   zend_restore_error_handling(&error_handling);
592:
593:   if (EG(exception)) {
594:     zend_object_iterator *sub_iter;
595:
596:     while (intern->level >= 0) {
597:       sub_iter = intern->iterators[intern->level].iterator;
598:       zend_iterator_dtor(sub_iter);
599:       zval_ptr_dtor(&intern->iterators[intern->level--].zobject);
600:     }
601:     efree(intern->iterators);
602:     intern->iterators = NULL;
603:   }
604: }
605:
606: /* {{{ proto void RecursiveIteratorIterator::__construct(RecursiveIterator|IteratorAggregate it [, int mode = RIT_LEAVES_ONLY [, int flags = 0]]) throw
s InvalidArgumentException
607:    Creates a RecursiveIteratorIterator from a RecursiveIterator. */
608: SPL_METHOD(RecursiveIteratorIterator, __construct)
609: {
610:   spl_recursive_it_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveIteratorIterator, zend_ce_iterator, RIT_RecursiveIteratorIterator);
611: } /* }}} */
612:
613: /* {{{ proto void RecursiveIteratorIterator::rewind()
614:    Rewind the iterator to the first element of the top level inner iterator. */
615: SPL_METHOD(RecursiveIteratorIterator, rewind)
616: {
617:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
618:
619:   if (zend_parse_parameters_none() == FAILURE) {
620:     return;
621:   }
622:
623:   spl_recursive_it_rewind_ex(object, getThis());
624: } /* }}} */
625:
626: /* {{{ proto bool RecursiveIteratorIterator::valid()
627:    Check whether the current position is valid */
628: SPL_METHOD(RecursiveIteratorIterator, valid)
629: {
630:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
631:
632:   if (zend_parse_parameters_none() == FAILURE) {
633:     return;
634:   }
635:
636:   RETURN_BOOL(spl_recursive_it_valid_ex(object, getThis()) == SUCCESS);
637: } /* }}} */
638:
639: /* {{{ proto mixed RecursiveIteratorIterator::key()
640:    Access the current key */
641: SPL_METHOD(RecursiveIteratorIterator, key)
642: {
643:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
644:   zend_object_iterator      *iterator;
645:
646:   if (zend_parse_parameters_none() == FAILURE) {
647:     return;
648:   }
649:
650:   SPL_FETCH_SUB_ITERATOR(iterator, object);
651:
652:   if (iterator->funcs->get_current_key) {
653:     iterator->funcs->get_current_key(iterator, return_value);
654:   } else {
655:     RETURN_NULL();
656:   }
657: } /* }}} */
658:
659: /* {{{ proto mixed RecursiveIteratorIterator::current()
660:    Access the current element value */
661: SPL_METHOD(RecursiveIteratorIterator, current)
662: {
663:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
664:   zend_object_iterator      *iterator;
665:   zval                      *data;
666:
667:   if (zend_parse_parameters_none() == FAILURE) {
668:     return;
669:   }
670:
671:   SPL_FETCH_SUB_ITERATOR(iterator, object);
672:
673:   data = iterator->funcs->get_current_data(iterator);
674:   if (data) {
675:     ZVAL_DEREF(data);
676:     ZVAL_COPY(return_value, data);
677:   }
678: } /* }}} */
679:
680: /* {{{ proto void RecursiveIteratorIterator::next()
681:    Move forward to the next element */
682: SPL_METHOD(RecursiveIteratorIterator, next)
683: {
684:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
685:
686:   if (zend_parse_parameters_none() == FAILURE) {
687:     return;
688:   }
689:
690:   spl_recursive_it_move_forward_ex(object, getThis());
691: } /* }}} */
692:
693: /* {{{ proto int RecursiveIteratorIterator::getDepth()
694:    Get the current depth of the recursive iteration */
695: SPL_METHOD(RecursiveIteratorIterator, getDepth)
696: {
697:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
698:
699:   if (zend_parse_parameters_none() == FAILURE) {
700:     return;
701:   }
702:
703:   RETURN_LONG(object->level);
704: } /* }}} */
705:
706: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::getSubIterator([int level])
707:    The current active sub iterator or the iterator at specified level */
708: SPL_METHOD(RecursiveIteratorIterator, getSubIterator)
709: {
710:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
711:   zend_long  level = object->level;
712:   zval *value;
713:
714:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "|l", &level) == FAILURE) {
715:     return;
716:   }
717:   if (level < 0 || level > object->level) {
718:     RETURN_NULL();
719:   }
720:
721:   if(!object->iterators) {
722:     zend_throw_exception_ex(spl_ce_LogicException, 0,
723:       "The object is in an invalid state as the parent constructor was not called");
724:     return;
725:   }
726:
727:   value = &object->iterators[level].zobject;
728:   ZVAL_DEREF(value);
729:   ZVAL_COPY(return_value, value);
730: } /* }}} */
731:
732: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::getInnerIterator()
733:    The current active sub iterator */
734: SPL_METHOD(RecursiveIteratorIterator, getInnerIterator)
735: {
736:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
737:   zval      *zobject;
738:
739:   if (zend_parse_parameters_none() == FAILURE) {
740:     return;
741:   }
742:
743:   SPL_FETCH_SUB_ELEMENT_ADDR(zobject, object, zobject);
744:
745:   ZVAL_DEREF(zobject);
746:   ZVAL_COPY(return_value, zobject);
747: } /* }}} */
```

```
748:
749: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::beginIteration()
750:    Called when iteration begins (after first rewind() call) */
751: SPL_METHOD(RecursiveIteratorIterator, beginIteration)
752: {
753:   if (zend_parse_parameters_none() == FAILURE) {
754:     return;
755:   }
756:   /* nothing to do */
757: } /* }}} */
758:
759: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::endIteration()
760:    Called when iteration ends (when valid() first returns false */
761: SPL_METHOD(RecursiveIteratorIterator, endIteration)
762: {
763:   if (zend_parse_parameters_none() == FAILURE) {
764:     return;
765:   }
766:   /* nothing to do */
767: } /* }}} */
768:
769: /* {{{ proto bool RecursiveIteratorIterator::callHasChildren()
770:    Called for each element to test whether it has children */
771: SPL_METHOD(RecursiveIteratorIterator, callHasChildren)
772: {
773:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
774:   zend_class_entry *ce;
775:   zval *zobject;
776:
777:   if (zend_parse_parameters_none() == FAILURE) {
778:     return;
779:   }
780:
781:   if (!object->iterators) {
782:     RETURN_NULL();
783:   }
784:
785:   SPL_FETCH_SUB_ELEMENT(ce, object, ce);
786:
787:   zobject = &object->iterators[object->level].zobject;
788:   if (Z_TYPE_P(zobject) == IS_UNDEF) {
789:     RETURN_FALSE;
790:   } else {
791:     zend_call_method_with_0_params(zobject, ce, NULL, "haschildren", return_value);
792:     if (Z_TYPE_P(return_value) == IS_UNDEF) {
793:       RETURN_FALSE;
794:     }
795:   }
796: } /* }}} */
797:
798: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::callGetChildren()
799:    Return children of current element */
800: SPL_METHOD(RecursiveIteratorIterator, callGetChildren)
801: {
802:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
803:   zend_class_entry *ce;
804:   zval *zobject;
805:
806:   if (zend_parse_parameters_none() == FAILURE) {
807:     return;
808:   }
809:
810:   SPL_FETCH_SUB_ELEMENT(ce, object, ce);
811:
812:   zobject = &object->iterators[object->level].zobject;
813:   if (Z_TYPE_P(zobject) == IS_UNDEF) {
814:     return;
815:   } else {
816:     zend_call_method_with_0_params(zobject, ce, NULL, "getchildren", return_value);
817:     if (Z_TYPE_P(return_value) == IS_UNDEF) {
818:       RETURN_NULL();
819:     }
820:   }
821: } /* }}} */
822:
823: /* {{{ proto void RecursiveIteratorIterator::beginChildren()
824:    Called when recursing one level down */
825: SPL_METHOD(RecursiveIteratorIterator, beginChildren)
826: {
827:   if (zend_parse_parameters_none() == FAILURE) {
828:     return;
829:   }
830:   /* nothing to do */
831: } /* }}} */
832:
833: /* {{{ proto void RecursiveIteratorIterator::endChildren()
834:    Called when end recursing one level */
835: SPL_METHOD(RecursiveIteratorIterator, endChildren)
836: {
837:   if (zend_parse_parameters_none() == FAILURE) {
838:     return;
839:   }
840:   /* nothing to do */
841: } /* }}} */
842:
843: /* {{{ proto void RecursiveIteratorIterator::nextElement()
844:    Called when the next element is available */
845: SPL_METHOD(RecursiveIteratorIterator, nextElement)
846: {
847:   if (zend_parse_parameters_none() == FAILURE) {
848:     return;
849:   }
850:   /* nothing to do */
851: } /* }}} */
852:
853: /* {{{ proto void RecursiveIteratorIterator::setMaxDepth([$max_depth = -1])
854:    Set the maximum allowed depth (or any depth if pmax_depth = -1] */
855: SPL_METHOD(RecursiveIteratorIterator, setMaxDepth)
856: {
857:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
858:   zend_long  max_depth = -1;
859:
860:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "|l", &max_depth) == FAILURE) {
861:     return;
862:   }
863:   if (max_depth < -1) {
864:     zend_throw_exception(spl_ce_OutOfRangeException, "Parameter max_depth must be >= -1", 0);
865:     return;
866:   } else if (max_depth > INT_MAX) {
867:     max_depth = INT_MAX;
868:   }
869:
870:   object->max_depth = (int)max_depth;
871: } /* }}} */
872:
873: /* {{{ proto int|false RecursiveIteratorIterator::getMaxDepth()
874:    Return the maximum accepted depth or false if any depth is allowed */
875: SPL_METHOD(RecursiveIteratorIterator, getMaxDepth)
876: {
877:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
878:
879:   if (zend_parse_parameters_none() == FAILURE) {
880:     return;
881:   }
882:
883:   if (object->max_depth == -1) {
884:     RETURN_FALSE;
885:   } else {
886:     RETURN_LONG(object->max_depth);
887:   }
888: } /* }}} */
889:
890: static union _zend_function *spl_recursive_it_get_method(zend_object **zobject, zend_string *method, const zval *key)
891: {
892:   union _zend_function    *function_handler;
893:   spl_recursive_it_object *object = spl_recursive_it_from_obj(*zobject);
894:   zend_long                level = object->level;
895:   zval                    *zobj;
896:
897:   if (!object->iterators) {
898:     php_error_docref(NULL, E_ERROR, "The %s instance wasn't initialized properly", ZSTR_VAL((*zobject)->ce->name));
899:   }
900:   zobj = &object->iterators[level].zobject;
901:
902:   function_handler = std_object_handlers.get_method(zobject, method, key);
903:   if (!function_handler) {
904:     if ((function_handler = zend_hash_find_ptr(&Z_OBJCE_P(zobj)->function_table, method)) == NULL) {
905:       if (Z_OBJ_HT_P(zobj)->get_method) {
906:         *zobject = Z_OBJ_P(zobj);
907:         function_handler = (*zobject)->handlers->get_method(zobject, method, key);
908:       }
909:     } else {
910:       *zobject = Z_OBJ_P(zobj);
911:     }
912:   }
913:   return function_handler;
914: }
915:
916: /* {{{ spl_RecursiveIteratorIterator_dtor */
917: static void spl_RecursiveIteratorIterator_dtor(zend_object *_object)
918: {
919:   spl_recursive_it_object *object = spl_recursive_it_from_obj(_object);
920:   zend_object_iterator *sub_iter;
921:
922:   /* call standard dtor */
923:   zend_objects_destroy_object(_object);
924:
925:   if (object->iterators) {
926:     while (object->level >= 0) {
927:       sub_iter = object->iterators[object->level].iterator;
928:       zend_iterator_dtor(sub_iter);
929:       zval_ptr_dtor(&object->iterators[object->level--].zobject);
930:     }
931:     efree(object->iterators);
932:     object->iterators = NULL;
933:   }
934: }
935: /* }}} */
936:
937: /* {{{ spl_RecursiveIteratorIterator_free_storage */
938: static void spl_RecursiveIteratorIterator_free_storage(zend_object *_object)
939: {
940:   spl_recursive_it_object *object = spl_recursive_it_from_obj(_object);
941:
942:   if (object->iterators) {
943:     efree(object->iterators);
944:     object->iterators = NULL;
945:     object->level     = 0;
946:   }
947:
948:   zend_object_std_dtor(&object->std);
949:   smart_str_free(&object->prefix[0]);
950:   smart_str_free(&object->prefix[1]);
951:   smart_str_free(&object->prefix[2]);
952:   smart_str_free(&object->prefix[3]);
953:   smart_str_free(&object->prefix[4]);
954:   smart_str_free(&object->prefix[5]);
955:
956:   smart_str_free(&object->postfix[0]);
957: }
958: /* }}} */
959:
960: /* {{{ spl_RecursiveIteratorIterator_new_ex */
961: static zend_object *spl_RecursiveIteratorIterator_new_ex(zend_class_entry *class_type, int init_prefix)
962: {
963:   spl_recursive_it_object *intern;
964:
965:   intern = zend_object_alloc(sizeof(spl_recursive_it_object), class_type);
966:
967:   if (init_prefix) {
968:     smart_str_appendl(&intern->prefix[0], "",    0);
969:     smart_str_appendl(&intern->prefix[1], "| ",  2);
970:     smart_str_appendl(&intern->prefix[2], "  ",  2);
971:     smart_str_appendl(&intern->prefix[3], "|-",  2);
972:     smart_str_appendl(&intern->prefix[4], "\\-", 2);
973:     smart_str_appendl(&intern->prefix[5], "",    0);
974:
975:     smart_str_appendl(&intern->postfix[0], "",    0);
976:   }
977:
978:   zend_object_std_init(&intern->std, class_type);
979:   object_properties_init(&intern->std, class_type);
980:
981:   intern->std.handlers = &spl_handlers_rec_it_it;
982:   return &intern->std;
983: }
984: /* }}} */
985:
986: /* {{{ spl_RecursiveIteratorIterator_new */
987: static zend_object *spl_RecursiveIteratorIterator_new(zend_class_entry *class_type)
988: {
989:   return spl_RecursiveIteratorIterator_new_ex(class_type, 0);
990: }
991: /* }}} */
992:
993: /* {{{ spl_RecursiveTreeIterator_new */
994: static zend_object *spl_RecursiveTreeIterator_new(zend_class_entry *class_type)
995: {
996:   return spl_RecursiveIteratorIterator_new_ex(class_type, 1);
997: }
998: /* }}} */
999:
1000: ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_it___construct, 0, 0, 1)
1001:   ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
1002:   ZEND_ARG_INFO(0, mode)
1003:   ZEND_ARG_INFO(0, flags)
1004: ZEND_END_ARG_INFO();
1005:
1006: ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_it_getSubIterator, 0, 0, 0)
1007:   ZEND_ARG_INFO(0, level)
1008: ZEND_END_ARG_INFO();
1009:
1010: ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_it_setMaxDepth, 0, 0, 0)
1011:   ZEND_ARG_INFO(0, max_depth)
1012: ZEND_END_ARG_INFO();
1013:
1014: static const zend_function_entry spl_funcs_RecursiveIteratorIterator[] = {
1015:   SPL_ME(RecursiveIteratorIterator, __construct,       arginfo_recursive_it___construct,    ZEND_ACC_PUBLIC)
1016:   SPL_ME(RecursiveIteratorIterator, rewind,            arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1017:   SPL_ME(RecursiveIteratorIterator, valid,             arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1018:   SPL_ME(RecursiveIteratorIterator, key,               arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1019:   SPL_ME(RecursiveIteratorIterator, current,           arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1020:   SPL_ME(RecursiveIteratorIterator, next,              arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1021:   SPL_ME(RecursiveIteratorIterator, getDepth,         arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1022:   SPL_ME(RecursiveIteratorIterator, getSubIterator,    arginfo_recursive_it_getSubIterator, ZEND_ACC_PUBLIC)
1023:   SPL_ME(RecursiveIteratorIterator, getInnerIterator,  arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1024:   SPL_ME(RecursiveIteratorIterator, beginIteration,    arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1025:   SPL_ME(RecursiveIteratorIterator, endIteration,      arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1026:   SPL_ME(RecursiveIteratorIterator, callHasChildren,   arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1027:   SPL_ME(RecursiveIteratorIterator, callGetChildren,   arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1028:   SPL_ME(RecursiveIteratorIterator, beginChildren,     arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1029:   SPL_ME(RecursiveIteratorIterator, endChildren,       arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1030:   SPL_ME(RecursiveIteratorIterator, nextElement,       arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1031:   SPL_ME(RecursiveIteratorIterator, setMaxDepth,       arginfo_recursive_it_setMaxDepth,    ZEND_ACC_PUBLIC)
1032:   SPL_ME(RecursiveIteratorIterator, getMaxDepth,       arginfo_recursive_it_void,           ZEND_ACC_PUBLIC)
1033:   PHP_FE_END
1034: };
1035:
1036: static void spl_recursive_tree_iterator_get_prefix(spl_recursive_it_object *object, zval *return_value)
1037: {
1038:   smart_str  str = {0};
1039:   zval       has_next;
1040:   int        level;
1041:
1042:   smart_str_appendl(&str, ZSTR_VAL(object->prefix[0].s), ZSTR_LEN(object->prefix[0].s));
1043:
1044:   for (level = 0; level < object->level; ++level) {
1045:     zend_call_method_with_0_params(&object->iterators[level].zobject, object->iterators[level].ce, NULL, "hasnext", &has_next);
1046:     if (Z_TYPE(has_next) != IS_UNDEF) {
1047:       if (Z_TYPE(has_next) == IS_TRUE) {
1048:         smart_str_appendl(&str, ZSTR_VAL(object->prefix[1].s), ZSTR_LEN(object->prefix[1].s));
1049:       } else {
1050:         smart_str_appendl(&str, ZSTR_VAL(object->prefix[2].s), ZSTR_LEN(object->prefix[2].s));
1051:       }
1052:       zval_ptr_dtor(&has_next);
1053:     }
1054:   }
1055:   zend_call_method_with_0_params(&object->iterators[level].zobject, object->iterators[level].ce, NULL, "hasnext", &has_next);
1056:   if (Z_TYPE(has_next) != IS_UNDEF) {
1057:     if (Z_TYPE(has_next) == IS_TRUE) {
1058:       smart_str_appendl(&str, ZSTR_VAL(object->prefix[3].s), ZSTR_LEN(object->prefix[3].s));
1059:     } else {
1060:       smart_str_appendl(&str, ZSTR_VAL(object->prefix[4].s), ZSTR_LEN(object->prefix[4].s));
1061:     }
1062:     zval_ptr_dtor(&has_next);
1063:   }
1064:
1065:   smart_str_appendl(&str, ZSTR_VAL(object->prefix[5].s), ZSTR_LEN(object->prefix[5].s));
1066:   smart_str_0(&str);
1067:
1068:   RETURN_NEW_STR(str.s);
1069: }
1070:
1071: static void spl_recursive_tree_iterator_get_entry(spl_recursive_it_object *object, zval *return_value)
1072: {
1073:   zend_object_iterator      *iterator = object->iterators[object->level].iterator;
1074:   zval                      *data;
1075:   zend_error_handling        error_handling;
1076:
1077:   data = iterator->funcs->get_current_data(iterator);
1078:
1079:   /* Replace exception handling so the catchable fatal error that is thrown when a class
1080:    * without __toString is converted to string is converted into an exception. */
1081:   zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);
1082:   if (data) {
1083:     ZVAL_DEREF(data);
1084:     if (Z_TYPE_P(data) == IS_ARRAY) {
1085:       ZVAL_STRINGL(return_value, "Array", sizeof("Array")-1);
1086:     } else {
1087:       ZVAL_COPY(return_value, data);
1088:       convert_to_string(return_value);
1089:     }
1090:   }
1091:   zend_restore_error_handling(&error_handling);
1092: }
1093:
1094: static void spl_recursive_tree_iterator_get_postfix(spl_recursive_it_object *object, zval *return_value)
1095: {
1096:   RETVAL_STR(object->postfix[0].s);
1097:   Z_ADDREF_P(return_value);
1098: }
1099:
1100: /* {{{ proto void RecursiveTreeIterator::__construct(RecursiveIterator|IteratorAggregate it [, int flags = RTIT_BYPASS_KEY [, int cit_flags = CIT_CATCH
_GET_CHILD [, mode = RIT_SELF_FIRST ]]]) throws InvalidArgumentException
1101:    RecursiveIteratorIterator to generate ASCII graphic trees for the entries in a RecursiveIterator */
1102: SPL_METHOD(RecursiveTreeIterator, __construct)
1103: {
1104:   spl_recursive_it_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveTreeIterator, zend_ce_iterator, RIT_RecursiveTreeIterator);
1105: } /* }}} */
1106:
1107: /* {{{ proto void RecursiveTreeIterator::setPrefixPart(int part, string prefix) throws OutOfRangeException
1108:    Sets prefix parts as used in getPrefix() */
1109: SPL_METHOD(RecursiveTreeIterator, setPrefixPart)
1110: {
1111:   zend_long  part;
1112:   char* prefix;
1113:   size_t   prefix_len;
1114:   spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
1115:
1116:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "ls", &part, &prefix, &prefix_len) == FAILURE) {
1117:     return;
1118:   }
1119:
1120:   if (0 > part || part > 5) {
1121:     zend_throw_exception_ex(spl_ce_OutOfRangeException, 0, "Use RecursiveTreeIterator::PREFIX_* constant");
1122:     return;
```

```
1123:  }
1124:
1125:  smart_str_free(&object->prefix[part]);
1126:  smart_str_appendl(&object->prefix[part], prefix, prefix_len);
1127: } /* }}} */
1128:
1129: /* {{{ proto string RecursiveTreeIterator::getPrefix()
1130:    Returns the string to place in front of current element */
1131: SPL_METHOD(RecursiveTreeIterator, getPrefix)
1132: {
1133:  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
1134:
1135:  if (zend_parse_parameters_none() == FAILURE) {
1136:    return;
1137:  }
1138:
1139:  if(!object->iterators) {
1140:    zend_throw_exception_ex(spl_ce_LogicException, 0,
1141:      "The object is in an invalid state as the parent constructor was not called");
1142:    return;
1143:  }
1144:
1145:  spl_recursive_tree_iterator_get_prefix(object, return_value);
1146: } /* }}} */
1147:
1148: /* {{{ proto void RecursiveTreeIterator::setPostfix(string prefix)
1149:    Sets postfix as used in getPostfix() */
1150: SPL_METHOD(RecursiveTreeIterator, setPostfix)
1151: {
1152:  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
1153:  char* postfix;
1154:  size_t   postfix_len;
1155:
1156:  if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &postfix, &postfix_len) == FAILURE) {
1157:    return;
1158:  }
1159:
1160:  smart_str_free(&object->postfix[0]);
1161:  smart_str_appendl(&object->postfix[0], postfix, postfix_len);
1162: } /* }}} */
1163:
1164: /* {{{ proto string RecursiveTreeIterator::getEntry()
1165:    Returns the string presentation built for current element */
1166: SPL_METHOD(RecursiveTreeIterator, getEntry)
1167: {
1168:  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
1169:
1170:  if (zend_parse_parameters_none() == FAILURE) {
1171:    return;
1172:  }
1173:
1174:  if(!object->iterators) {
1175:    zend_throw_exception_ex(spl_ce_LogicException, 0,
1176:      "The object is in an invalid state as the parent constructor was not called");
1177:    return;
1178:  }
1179:
1180:  spl_recursive_tree_iterator_get_entry(object, return_value);
1181: } /* }}} */
1182:
1183: /* {{{ proto string RecursiveTreeIterator::getPostfix()
1184:    Returns the string to place after the current element */
1185: SPL_METHOD(RecursiveTreeIterator, getPostfix)
1186: {
1187:  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
1188:
1189:  if (zend_parse_parameters_none() == FAILURE) {
1190:    return;
1191:  }
1192:
1193:  if(!object->iterators) {
1194:    zend_throw_exception_ex(spl_ce_LogicException, 0,
1195:      "The object is in an invalid state as the parent constructor was not called");
1196:    return;
1197:  }
1198:
1199:  spl_recursive_tree_iterator_get_postfix(object, return_value);
1200: } /* }}} */
1201:
1202: /* {{{ proto mixed RecursiveTreeIterator::current()
1203:    Returns the current element prefixed and postfixed */
1204: SPL_METHOD(RecursiveTreeIterator, current)
1205: {
1206:  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
1207:  zval                      prefix, entry, postfix;
1208:  char                      *ptr;
1209:  zend_string               *str;
1210:
1211:  if (zend_parse_parameters_none() == FAILURE) {
1212:    return;
1213:  }
1214:
1215:  if(!object->iterators) {
1216:    zend_throw_exception_ex(spl_ce_LogicException, 0,
1217:      "The object is in an invalid state as the parent constructor was not called");
1218:    return;
1219:  }
1220:
1221:  if (object->flags & RTIT_BYPASS_CURRENT) {
1222:    zend_object_iterator      *iterator = object->iterators[object->level].iterator;
1223:    zval                      *data;
1224:
1225:        SPL_FETCH_SUB_ITERATOR(iterator, object);
1226:    data = iterator->funcs->get_current_data(iterator);
1227:    if (data) {
1228:      ZVAL_DEREF(data);
1229:      ZVAL_COPY(return_value, data);
1230:      return;
1231:    } else {
1232:      RETURN_NULL();
1233:    }
1234:  }
1235:
1236:  ZVAL_NULL(&prefix);
1237:  ZVAL_NULL(&entry);
1238:  spl_recursive_tree_iterator_get_prefix(object, &prefix);
1239:  spl_recursive_tree_iterator_get_entry(object, &entry);
1240:  if (Z_TYPE(entry) != IS_STRING) {
1241:    zval_ptr_dtor(&prefix);
1242:    zval_ptr_dtor(&entry);
1243:    RETURN_NULL();
1244:  }
1245:  spl_recursive_tree_iterator_get_postfix(object, &postfix);
1246:
1247:  str = zend_string_alloc(Z_STRLEN(prefix) + Z_STRLEN(entry) + Z_STRLEN(postfix), 0);
1248:  ptr = ZSTR_VAL(str);
1249:
1250:  memcpy(ptr, Z_STRVAL(prefix), Z_STRLEN(prefix));
1251:  ptr += Z_STRLEN(prefix);
1252:  memcpy(ptr, Z_STRVAL(entry), Z_STRLEN(entry));
1253:  ptr += Z_STRLEN(entry);
1254:  memcpy(ptr, Z_STRVAL(postfix), Z_STRLEN(postfix));
1255:  ptr += Z_STRLEN(postfix);
1256:  *ptr = 0;
1257:
1258:  zval_ptr_dtor(&prefix);
1259:  zval_ptr_dtor(&entry);
1260:  zval_ptr_dtor(&postfix);
1261:
1262:  RETURN_NEW_STR(str);
1263: } /* }}} */
1264:
1265: /* {{{ proto mixed RecursiveTreeIterator::key()
1266:    Returns the current key prefixed and postfixed */
1267: SPL_METHOD(RecursiveTreeIterator, key)
1268: {
1269:  spl_recursive_it_object   *object = Z_SPLRECURSIVE_IT_P(getThis());
1270:  zend_object_iterator      *iterator;
1271:  zval                      prefix, key, postfix, key_copy;
1272:  char                      *ptr;
1273:  zend_string               *str;
1274:
1275:  if (zend_parse_parameters_none() == FAILURE) {
1276:    return;
1277:  }
1278:
1279:  SPL_FETCH_SUB_ITERATOR(iterator, object);
1280:
1281:  if (iterator->funcs->get_current_key) {
1282:    iterator->funcs->get_current_key(iterator, &key);
1283:  } else {
1284:    ZVAL_NULL(&key);
1285:  }
1286:
1287:  if (object->flags & RTIT_BYPASS_KEY) {
1288:    RETVAL_ZVAL(&key, 1, 1);
1289:    return;
1290:  }
1291:
1292:  if (Z_TYPE(key) != IS_STRING) {
1293:    if (zend_make_printable_zval(&key, &key_copy)) {
1294:      key = key_copy;
1295:    }
1296:  }
1297:
1298:  spl_recursive_tree_iterator_get_prefix(object, &prefix);
1299:  spl_recursive_tree_iterator_get_postfix(object, &postfix);
1300:
1301:  str = zend_string_alloc(Z_STRLEN(prefix) + Z_STRLEN(key) + Z_STRLEN(postfix), 0);
1302:  ptr = ZSTR_VAL(str);
1303:
1304:  memcpy(ptr, Z_STRVAL(prefix), Z_STRLEN(prefix));
1305:  ptr += Z_STRLEN(prefix);
1306:  memcpy(ptr, Z_STRVAL(key), Z_STRLEN(key));
1307:  ptr += Z_STRLEN(key);
1308:  memcpy(ptr, Z_STRVAL(postfix), Z_STRLEN(postfix));
1309:  ptr += Z_STRLEN(postfix);
1310:  *ptr = 0;
1311:
1312:  zval_ptr_dtor(&prefix);
1313:  zval_ptr_dtor(&key);
1314:  zval_ptr_dtor(&postfix);
1315:
1316:  RETURN_NEW_STR(str);
1317: } /* }}} */
1318:
1319: ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_tree_it___construct, 0, 0, 1)
1320:  ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
1321:  ZEND_ARG_INFO(0, flags)
1322:  ZEND_ARG_INFO(0, caching_it_flags)
1323:  ZEND_ARG_INFO(0, mode)
1324: ZEND_END_ARG_INFO();
1325:
1326: ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_tree_it_setPrefixPart, 0, 0, 2)
1327:  ZEND_ARG_INFO(0, part)
1328:  ZEND_ARG_INFO(0, value)
1329: ZEND_END_ARG_INFO();
1330:
1331: ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_tree_it_setPostfix, 0, 0, 1)
1332:  ZEND_ARG_INFO(0, postfix)
1333: ZEND_END_ARG_INFO();
1334:
1335: static const zend_function_entry spl_funcs_RecursiveTreeIterator[] = {
1336:  SPL_ME(RecursiveTreeIterator,     __construct,       arginfo_recursive_tree_it___construct,   ZEND_ACC_PUBLIC)
1337:  SPL_ME(RecursiveIteratorIterator, rewind,            arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1338:  SPL_ME(RecursiveIteratorIterator, valid,             arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1339:  SPL_ME(RecursiveTreeIterator,     key,               arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1340:  SPL_ME(RecursiveIteratorIterator, current,           arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1341:  SPL_ME(RecursiveIteratorIterator, next,              arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1342:  SPL_ME(RecursiveIteratorIterator, beginIteration,    arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1343:  SPL_ME(RecursiveIteratorIterator, endIteration,      arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1344:  SPL_ME(RecursiveIteratorIterator, callHasChildren,   arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1345:  SPL_ME(RecursiveIteratorIterator, callGetChildren,   arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1346:  SPL_ME(RecursiveIteratorIterator, beginChildren,     arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1347:  SPL_ME(RecursiveIteratorIterator, endChildren,       arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1348:  SPL_ME(RecursiveIteratorIterator, nextElement,       arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1349:  SPL_ME(RecursiveTreeIterator,     getPrefix,         arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1350:  SPL_ME(RecursiveTreeIterator,     setPrefixPart,     arginfo_recursive_tree_it_setPrefixPart, ZEND_ACC_PUBLIC)
1351:  SPL_ME(RecursiveTreeIterator,     getEntry,          arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1352:  SPL_ME(RecursiveTreeIterator,     setPostfix,        arginfo_recursive_tree_it_setPostfix,              ZEND_ACC_PUBLIC)
1353:  SPL_ME(RecursiveTreeIterator,     getPostfix,        arginfo_recursive_it_void,               ZEND_ACC_PUBLIC)
1354:  PHP_FE_END
1355: };
1356:
1357: #if MBO_0
1358: static int spl_dual_it_gets_implemented(zend_class_entry *interface, zend_class_entry *class_type)
1359: {
1360:  class_type->iterator_funcs.zf_valid = NULL;
1361:  class_type->iterator_funcs.zf_current = NULL;
1362:  class_type->iterator_funcs.zf_key = NULL;
1363:  class_type->iterator_funcs.zf_next = NULL;
1364:  class_type->iterator_funcs.zf_rewind = NULL;
1365:  if (!class_type->iterator_funcs.funcs) {
1366:    class_type->iterator_funcs.funcs = &zend_interface_iterator_funcs_iterator;
1367:  }
1368:
1369:  return SUCCESS;
1370: }
1371: #endif
1372:
1373: static union _zend_function *spl_dual_it_get_method(zend_object **object, zend_string *method, const zval *key)
1374: {
1375:  union _zend_function *function_handler;
1376:  spl_dual_it_object   *intern;
1377:
1378:  intern = spl_dual_it_from_obj(*object);
1379:
1380:  function_handler = std_object_handlers.get_method(object, method, key);
1381:  if (!function_handler && intern->inner.ce) {
1382:    if ((function_handler = zend_hash_find_ptr(&intern->inner.ce->function_table, method)) == NULL) {
1383:      if (Z_OBJ_HT(intern->inner.zobject)->get_method) {
1384:        *object = Z_OBJ(intern->inner.zobject);
1385:        function_handler = (*object)->handlers->get_method(object, method, key);
1386:      }
1387:    } else {
1388:      *object = Z_OBJ(intern->inner.zobject);
1389:    }
1390:  }
1391:  return function_handler;
1392: }
1393:
1394: #if MBO_0
1395: int spl_dual_it_call_method(char *method, INTERNAL_FUNCTION_PARAMETERS)
1396: {
1397:  zval ***func_params, func;
1398:  zval retval;
1399:  int arg_count;
1400:  int current = 0;
1401:  int success;
1402:  void **p;
1403:  spl_dual_it_object   *intern;
1404:
1405:  intern = Z_SPLDUAL_IT_P(getThis());
1406:
1407:  ZVAL_STRING(&func, method, 0);
1408:
1409:  p = EG(argument_stack).top_element-2;
1410:  arg_count = (zend_ulong) *p;
1411:
1412:  func_params = safe_emalloc(sizeof(zval **), arg_count, 0);
1413:
1414:  current = 0;
1415:  while (arg_count-- > 0) {
1416:    func_params[current] = (zval **) p - (arg_count-current);
1417:    current++;
1418:  }
1419:  arg_count = current; /* restore */
1420:
1421:  if (call_user_function_ex(EG(function_table), NULL, &func, &retval, arg_count, func_params, 0, NULL) == SUCCESS && Z_TYPE(retval) != IS_UNDEF) {
1422:    RETURN_ZVAL(&retval, 0, 0);
1423:
1424:    success = SUCCESS;
1425:  } else {
1426:    zend_throw_error(NULL, "Unable to call %s::%s()", intern->inner.ce->name, method);
1427:    success = FAILURE;
1428:  }
1429:
1430:  efree(func_params);
1431:  return success;
1432: }
1433: #endif
1434:
1435: #define SPL_CHECK_CTOR(intern, classname) \
1436:  if (intern->dit_type == DIT_Unknown) { \
1437:    zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "Classes derived from %s must call %s::__construct()", \
1438:      ZSTR_VAL((spl_ce_##classname)->name), ZSTR_VAL((spl_ce_##classname)->name)); \
1439:    return; \
1440:  }
1441:
1442: #define APPENDIT_CHECK_CTOR(intern) SPL_CHECK_CTOR(intern, AppendIterator)
1443:
1444: static inline int spl_dual_it_fetch(spl_dual_it_object *intern, int check_more);
1445:
1446: static inline int spl_cit_check_flags(zend_long flags)
1447: {
1448:  zend_long cnt = 0;
1449:
1450:  cnt += (flags & CIT_CALL_TOSTRING) ? 1 : 0;
1451:  cnt += (flags & CIT_TOSTRING_USE_KEY) ? 1 : 0;
1452:  cnt += (flags & CIT_TOSTRING_USE_CURRENT) ? 1 : 0;
1453:  cnt += (flags & CIT_TOSTRING_USE_INNER) ? 1 : 0;
1454:
1455:  return cnt <= 1 ? SUCCESS : FAILURE;
1456: }
1457:
1458: static spl_dual_it_object* spl_dual_it_construct(INTERNAL_FUNCTION_PARAMETERS, zend_class_entry *ce_base, zend_class_entry *ce_inner, dual_it_type dit_
type)
1459: {
1460:  zval                  *zobject, retval;
1461:  spl_dual_it_object   *intern;
1462:  zend_class_entry     *ce = NULL;
1463:  int                   inc_refcount = 1;
1464:  zend_error_handling   error_handling;
1465:
1466:  intern = Z_SPLDUAL_IT_P(getThis());
1467:
1468:  if (intern->dit_type != DIT_Unknown) {
1469:    zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s::getIterator() must be called exactly once per instance", ZSTR_VAL(ce_base->name));
1470:    return NULL;
1471:  }
1472:
1473:  intern->dit_type = dit_type;
1474:  switch (dit_type) {
1475:    case DIT_LimitIterator: {
1476:      intern->u.limit.offset = 0; /* start at beginning */
1477:      intern->u.limit.count = -1; /* get all */
1478:      if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "O|ll", &zobject, ce_inner, &intern->u.limit.offset, &intern->u.limit.count) == FAILURE) {
1479:        return NULL;
1480:      }
1481:      if (intern->u.limit.offset < 0) {
1482:        zend_throw_exception(spl_ce_OutOfRangeException, "Parameter offset must be >= 0", 0);
1483:        return NULL;
1484:      }
1485:      if (intern->u.limit.count < 0 && intern->u.limit.count != -1) {
1486:        zend_throw_exception(spl_ce_OutOfRangeException, "Parameter count must either be -1 or a value greater than or equal 0", 0);
1487:        return NULL;
1488:      }
1489:      break;
1490:    }
1491:    case DIT_CachingIterator:
1492:    case DIT_RecursiveCachingIterator: {
1493:      zend_long flags = CIT_CALL_TOSTRING;
1494:      if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "O|l", &zobject, ce_inner, &flags) == FAILURE) {
1495:        return NULL;
1496:      }
1497:      if (spl_cit_check_flags(flags) != SUCCESS) {
```

```
1498:          zend_throw_exception(spl_ce_InvalidArgumentException, "Flags must contain only one of CALL_TOSTRING, TOSTRING_USE_KEY, TOSTRING_USE_CURRENT, TO
STRING_USE_INNER", 0);
1499:          return NULL;
1500:        }
1501:        intern->u.caching.flags |= flags & CIT_PUBLIC;
1502:        array_init(&intern->u.caching.zcache);
1503:        break;
1504:      }
1505:      case DIT_IteratorIterator: {
1506:        zend_class_entry *ce_cast;
1507:        zend_string *class_name;
1508:
1509:        if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "O|S", &zobject, ce_inner, &class_name) == FAILURE) {
1510:          return NULL;
1511:        }
1512:        ce = Z_OBJCE_P(zobject);
1513:        if (!instanceof_function(ce, zend_ce_iterator)) {
1514:          if (ZEND_NUM_ARGS() > 1) {
1515:            if (!(ce_cast = zend_lookup_class(class_name))
1516:            || !instanceof_function(ce, ce_cast)
1517:            || !ce_cast->get_iterator
1518:            ) {
1519:              zend_throw_exception(spl_ce_LogicException, "Class to downcast to not found or not base class or does not implement Traversable", 0);
1520:              return NULL;
1521:            }
1522:            ce = ce_cast;
1523:          }
1524:          if (instanceof_function(ce, zend_ce_aggregate)) {
1525:            zend_call_method_with_0_params(zobject, ce, &ce->iterator_funcs.zf_new_iterator, "getiterator", &retval);
1526:            if (EG(exception)) {
1527:              zval_ptr_dtor(&retval);
1528:              return NULL;
1529:            }
1530:            if (Z_TYPE(retval) != IS_OBJECT || !instanceof_function(Z_OBJCE(retval), zend_ce_traversable)) {
1531:              zend_throw_exception_ex(spl_ce_LogicException, 0, "%s::getIterator() must return an object that implements Traversable", ZSTR_VAL(ce->name)
);
1532:              return NULL;
1533:            }
1534:            zobject = &retval;
1535:            ce = Z_OBJCE_P(zobject);
1536:            inc_refcount = 0;
1537:          }
1538:        }
1539:        break;
1540:      }
1541:      case DIT_AppendIterator:
1542:        zend_replace_error_handling(EH_THROW, spl_ce_InvalidArgumentException, &error_handling);
1543:        spl_instantiate(spl_ce_ArrayIterator, &intern->u.append.zarrayit);
1544:        zend_call_method_with_0_params(&intern->u.append.zarrayit, spl_ce_ArrayIterator, &spl_ce_ArrayIterator->constructor, "__construct", NULL);
1545:        intern->u.append.iterator = spl_ce_ArrayIterator->get_iterator(spl_ce_ArrayIterator, &intern->u.append.zarrayit, 0);
1546:        zend_restore_error_handling(&error_handling);
1547:        return intern;
1548: #if HAVE_PCRE || HAVE_BUNDLED_PCRE
1549:      case DIT_RegexIterator:
1550:      case DIT_RecursiveRegexIterator: {
1551:        zend_string *regex;
1552:        zend_long mode = REGIT_MODE_MATCH;
1553:
1554:        intern->u.regex.use_flags = ZEND_NUM_ARGS() >= 5;
1555:        intern->u.regex.flags = 0;
1556:        intern->u.regex.preg_flags = 0;
1557:        if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "OS|lll", &zobject, ce_inner, &regex, &mode, &intern->u.regex.flags, &intern->u.regex.preg_flags
) == FAILURE) {
1558:          return NULL;
1559:        }
1560:        if (mode < 0 || mode >= REGIT_MODE_MAX) {
1561:          zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "Illegal mode " ZEND_LONG_FMT, mode);
1562:          return NULL;
1563:        }
1564:        intern->u.regex.mode = mode;
1565:        intern->u.regex.regex = zend_string_copy(regex);
1566:
1567:        zend_replace_error_handling(EH_THROW, spl_ce_InvalidArgumentException, &error_handling);
1568:        intern->u.regex.pce = pcre_get_compiled_regex_cache(regex);
1569:        zend_restore_error_handling(&error_handling);
1570:
1571:        if (intern->u.regex.pce == NULL) {
1572:          /* pcre_get_compiled_regex_cache has already sent error */
1573:          return NULL;
1574:        }
1575:        php_pcre_pce_incref(intern->u.regex.pce);
1576:        break;
1577:      }
1578: #endif
1579:      case DIT_CallbackFilterIterator:
1580:      case DIT_RecursiveCallbackFilterIterator: {
1581:        _spl_cbfilter_it_intern *cfi = emalloc(sizeof(*cfi));
1582:        cfi->fci.object = NULL;
1583:        if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "Of", &zobject, ce_inner, &cfi->fci, &cfi->fcc) == FAILURE) {
1584:          efree(cfi);
1585:          return NULL;
1586:        }
1587:        Z_TRY_ADDREF(cfi->fci.function_name);
1588:        cfi->object = cfi->fcc.object;
1589:        if (cfi->object) GC_ADDREF(cfi->object);
1590:        intern->u.cbfilter = cfi;
1591:        break;
1592:      }
1593:      default:
1594:        if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "O", &zobject, ce_inner) == FAILURE) {
1595:          return NULL;
1596:        }
1597:        break;
1598:    }
1599:
1600:    if (inc_refcount) {
1601:      ZVAL_COPY(&intern->inner.zobject, zobject);
1602:    } else {
1603:      ZVAL_COPY_VALUE(&intern->inner.zobject, zobject);
1604:    }
1605:
1606:    intern->inner.ce = dit_type == DIT_IteratorIterator ? ce : Z_OBJCE_P(zobject);
1607:    intern->inner.object = Z_OBJ_P(zobject);
1608:    intern->inner.iterator = intern->inner.ce->get_iterator(intern->inner.ce, zobject, 0);
1609:
1610:    return intern;
1611: }
1612:
1613: /* {{{ proto void FilterIterator::__construct(Iterator it)
1614:    Create an Iterator from another iterator */
1615: SPL_METHOD(FilterIterator, __construct)
1616: {
1617:    spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_FilterIterator, zend_ce_iterator, DIT_FilterIterator);
1618: } /* }}} */
1619:
1620: /* {{{ proto void CallbackFilterIterator::__construct(Iterator it, callback func)
1621:    Create an Iterator from another iterator */
1622: SPL_METHOD(CallbackFilterIterator, __construct)
1623: {
1624:    spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_CallbackFilterIterator, zend_ce_iterator, DIT_CallbackFilterIterator);
1625: } /* }}} */
1626:
1627: /* {{{ proto Iterator FilterIterator::getInnerIterator()
1628:        proto Iterator CachingIterator::getInnerIterator()
1629:        proto Iterator LimitIterator::getInnerIterator()
1630:        proto Iterator ParentIterator::getInnerIterator()
1631:    Get the inner iterator */
1632: SPL_METHOD(dual_it, getInnerIterator)
1633: {
1634:    spl_dual_it_object   *intern;
1635:
1636:    if (zend_parse_parameters_none() == FAILURE) {
1637:      return;
1638:    }
1639:
1640:    SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1641:
1642:    if (!Z_ISUNDEF(intern->inner.zobject)) {
1643:      zval *value = &intern->inner.zobject;
1644:
1645:      ZVAL_DEREF(value);
1646:      ZVAL_COPY(return_value, value);
1647:    } else {
1648:      RETURN_NULL();
1649:    }
1650: } /* }}} */
1651:
1652: static inline void spl_dual_it_free(spl_dual_it_object *intern)
1653: {
1654:    if (intern->inner.iterator && intern->inner.iterator->funcs->invalidate_current) {
1655:      intern->inner.iterator->funcs->invalidate_current(intern->inner.iterator);
1656:    }
1657:    if (Z_TYPE(intern->current.data) != IS_UNDEF) {
1658:      zval_ptr_dtor(&intern->current.data);
1659:      ZVAL_UNDEF(&intern->current.data);
1660:    }
1661:    if (Z_TYPE(intern->current.key) != IS_UNDEF) {
1662:      zval_ptr_dtor(&intern->current.key);
1663:      ZVAL_UNDEF(&intern->current.key);
1664:    }
1665:    if (intern->dit_type == DIT_CachingIterator || intern->dit_type == DIT_RecursiveCachingIterator) {
1666:      if (Z_TYPE(intern->u.caching.zstr) != IS_UNDEF) {
1667:        zval_ptr_dtor(&intern->u.caching.zstr);
1668:        ZVAL_UNDEF(&intern->u.caching.zstr);
1669:      }
1670:      if (Z_TYPE(intern->u.caching.zchildren) != IS_UNDEF) {
1671:        zval_ptr_dtor(&intern->u.caching.zchildren);
1672:        ZVAL_UNDEF(&intern->u.caching.zchildren);
1673:      }
1674:    }
1675: }
1676:
1677: static inline void spl_dual_it_rewind(spl_dual_it_object *intern)
1678: {
1679:    spl_dual_it_free(intern);
1680:    intern->current.pos = 0;
1681:    if (intern->inner.iterator && intern->inner.iterator->funcs->rewind) {
1682:      intern->inner.iterator->funcs->rewind(intern->inner.iterator);
1683:    }
1684: }
1685:
1686: static inline int spl_dual_it_valid(spl_dual_it_object *intern)
1687: {
1688:    if (!intern->inner.iterator) {
1689:      return FAILURE;
1690:    }
1691:    /* FAILURE / SUCCESS */
1692:    return intern->inner.iterator->funcs->valid(intern->inner.iterator);
1693: }
1694:
1695: static inline int spl_dual_it_fetch(spl_dual_it_object *intern, int check_more)
1696: {
1697:    zval *data;
1698:
1699:    spl_dual_it_free(intern);
1700:    if (!check_more || spl_dual_it_valid(intern) == SUCCESS) {
1701:      data = intern->inner.iterator->funcs->get_current_data(intern->inner.iterator);
1702:      if (data) {
1703:        ZVAL_COPY(&intern->current.data, data);
1704:      }
1705:
1706:      if (intern->inner.iterator->funcs->get_current_key) {
1707:        intern->inner.iterator->funcs->get_current_key(intern->inner.iterator, &intern->current.key);
1708:        if (EG(exception)) {
1709:          zval_ptr_dtor(&intern->current.key);
1710:          ZVAL_UNDEF(&intern->current.key);
1711:        }
1712:      } else {
1713:        ZVAL_LONG(&intern->current.key, intern->current.pos);
1714:      }
1715:      return EG(exception) ? FAILURE : SUCCESS;
1716:    }
1717:    return FAILURE;
1718: }
1719:
1720: static inline void spl_dual_it_next(spl_dual_it_object *intern, int do_free)
1721: {
1722:    if (do_free) {
1723:      spl_dual_it_free(intern);
1724:    } else if (!intern->inner.iterator) {
1725:      zend_throw_error(NULL, "The inner constructor wasn't initialized with an iterator instance");
1726:      return;
1727:    }
1728:    intern->inner.iterator->funcs->move_forward(intern->inner.iterator);
1729:    intern->current.pos++;
1730: }
1731:
1732: /* {{{ proto void ParentIterator::rewind()
1733:        proto void IteratorIterator::rewind()
1734:    Rewind the iterator
1735:    */
1736: SPL_METHOD(dual_it, rewind)
1737: {
1738:    spl_dual_it_object   *intern;
1739:
1740:    if (zend_parse_parameters_none() == FAILURE) {
1741:      return;
1742:    }
1743:
1744:    SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1745:
1746:    spl_dual_it_rewind(intern);
1747:    spl_dual_it_fetch(intern, 1);
1748: } /* }}} */
1749:
1750: /* {{{ proto bool FilterIterator::valid()
1751:        proto bool ParentIterator::valid()
1752:        proto bool IteratorIterator::valid()
1753:        proto bool NoRewindIterator::valid()
1754:    Check whether the current element is valid */
1755: SPL_METHOD(dual_it, valid)
1756: {
1757:    spl_dual_it_object   *intern;
1758:
1759:    if (zend_parse_parameters_none() == FAILURE) {
1760:      return;
1761:    }
1762:
1763:    SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1764:
1765:    RETURN_BOOL(Z_TYPE(intern->current.data) != IS_UNDEF);
1766: } /* }}} */
1767:
1768: /* {{{ proto mixed FilterIterator::key()
1769:        proto mixed CachingIterator::key()
1770:        proto mixed LimitIterator::key()
1771:        proto mixed ParentIterator::key()
1772:        proto mixed IteratorIterator::key()
1773:        proto mixed NoRewindIterator::key()
1774:        proto mixed AppendIterator::key()
1775:    Get the current key */
1776: SPL_METHOD(dual_it, key)
1777: {
1778:    spl_dual_it_object   *intern;
1779:
1780:    if (zend_parse_parameters_none() == FAILURE) {
1781:      return;
1782:    }
1783:
1784:    SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1785:
1786:    if (Z_TYPE(intern->current.key) != IS_UNDEF) {
1787:      zval *value = &intern->current.key;
1788:
1789:      ZVAL_DEREF(value);
1790:      ZVAL_COPY(return_value, value);
1791:    } else {
1792:      RETURN_NULL();
1793:    }
1794: } /* }}} */
1795:
1796: /* {{{ proto mixed FilterIterator::current()
1797:        proto mixed CachingIterator::current()
1798:        proto mixed LimitIterator::current()
1799:        proto mixed ParentIterator::current()
1800:        proto mixed IteratorIterator::current()
1801:        proto mixed NoRewindIterator::current()
1802:    Get the current element value */
1803: SPL_METHOD(dual_it, current)
1804: {
1805:    spl_dual_it_object   *intern;
1806:
1807:    if (zend_parse_parameters_none() == FAILURE) {
1808:      return;
1809:    }
1810:
1811:    SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1812:
1813:    if (Z_TYPE(intern->current.data) != IS_UNDEF) {
1814:      zval *value = &intern->current.data;
1815:
1816:      ZVAL_DEREF(value);
1817:      ZVAL_COPY(return_value, value);
1818:    } else {
1819:      RETURN_NULL();
1820:    }
1821: } /* }}} */
1822:
1823: /* {{{ proto void ParentIterator::next()
1824:        proto void IteratorIterator::next()
1825:        proto void NoRewindIterator::next()
1826:    Move the iterator forward */
1827: SPL_METHOD(dual_it, next)
1828: {
1829:    spl_dual_it_object   *intern;
1830:
1831:    if (zend_parse_parameters_none() == FAILURE) {
1832:      return;
1833:    }
1834:
1835:    SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1836:
1837:    spl_dual_it_next(intern, 1);
1838:    spl_dual_it_fetch(intern, 1);
1839: } /* }}} */
1840:
1841: static inline void spl_filter_it_fetch(zval *zthis, spl_dual_it_object *intern)
1842: {
1843:    zval retval;
1844:
1845:    while (spl_dual_it_fetch(intern, 1) == SUCCESS) {
1846:      zend_call_method_with_0_params(zthis, intern->std.ce, NULL, "accept", &retval);
1847:      if (Z_TYPE(retval) != IS_UNDEF) {
1848:        if (zend_is_true(&retval)) {
1849:          zval_ptr_dtor(&retval);
1850:          return;
1851:        }
1852:        zval_ptr_dtor(&retval);
1853:      }
1854:      if (EG(exception)) {
1855:        return;
1856:      }
1857:      intern->inner.iterator->funcs->move_forward(intern->inner.iterator);
1858:    }
1859:    spl_dual_it_free(intern);
1860: }
1861:
1862: static inline void spl_filter_it_rewind(zval *zthis, spl_dual_it_object *intern)
1863: {
1864:    spl_dual_it_rewind(intern);
1865:    spl_filter_it_fetch(zthis, intern);
1866: }
1867:
1868: static inline void spl_filter_it_next(zval *zthis, spl_dual_it_object *intern)
1869: {
1870:    spl_dual_it_next(intern, 1);
```

```
1871:   spl_filter_it_fetch(&this, intern);
1872: }
1873:
1874: /* {{{ proto void FilterIterator::rewind()
1875:    Rewind the iterator */
1876: SPL_METHOD(FilterIterator, rewind)
1877: {
1878:   spl_dual_it_object   *intern;
1879:
1880:   if (zend_parse_parameters_none() == FAILURE) {
1881:     return;
1882:   }
1883:
1884:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1885:   spl_filter_it_rewind(getThis(), intern);
1886: } /* }}} */
1887:
1888: /* {{{ proto void FilterIterator::next()
1889:    Move the iterator forward */
1890: SPL_METHOD(FilterIterator, next)
1891: {
1892:   spl_dual_it_object   *intern;
1893:
1894:   if (zend_parse_parameters_none() == FAILURE) {
1895:     return;
1896:   }
1897:
1898:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1899:   spl_filter_it_next(getThis(), intern);
1900: } /* }}} */
1901:
1902: /* {{{ proto void RecursiveCallbackFilterIterator::__construct(RecursiveIterator it, callback func)
1903:    Create a RecursiveCallbackFilterIterator from a RecursiveIterator */
1904: SPL_METHOD(RecursiveCallbackFilterIterator, __construct)
1905: {
1906:   spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveCallbackFilterIterator, spl_ce_RecursiveIterator, DIT_RecursiveCallbackFilter
Iterator);
1907: } /* }}} */
1908:
1909:
1910: /* {{{ proto void RecursiveFilterIterator::__construct(RecursiveIterator it)
1911:    Create a RecursiveFilterIterator from a RecursiveIterator */
1912: SPL_METHOD(RecursiveFilterIterator, __construct)
1913: {
1914:   spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveFilterIterator, spl_ce_RecursiveIterator, DIT_RecursiveFilterIterator);
1915: } /* }}} */
1916:
1917: /* {{{ proto bool RecursiveFilterIterator::hasChildren()
1918:    Check whether the inner iterator's current element has children */
1919: SPL_METHOD(RecursiveFilterIterator, hasChildren)
1920: {
1921:   spl_dual_it_object   *intern;
1922:   zval                 retval;
1923:
1924:   if (zend_parse_parameters_none() == FAILURE) {
1925:     return;
1926:   }
1927:
1928:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1929:
1930:   zend_call_method_with_0_params(&intern->inner.zobject, intern->inner.ce, NULL, "haschildren", &retval);
1931:   if (Z_TYPE(retval) != IS_UNDEF) {
1932:     RETURN_ZVAL(&retval, 0, 1);
1933:   } else {
1934:     RETURN_FALSE;
1935:   }
1936: } /* }}} */
1937:
1938: /* {{{ proto RecursiveFilterIterator RecursiveFilterIterator::getChildren()
1939:    Return the inner iterator's children contained in a RecursiveFilterIterator */
1940: SPL_METHOD(RecursiveFilterIterator, getChildren)
1941: {
1942:   spl_dual_it_object   *intern;
1943:   zval                 retval;
1944:
1945:   if (zend_parse_parameters_none() == FAILURE) {
1946:     return;
1947:   }
1948:
1949:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1950:
1951:   zend_call_method_with_0_params(&intern->inner.zobject, intern->inner.ce, NULL, "getchildren", &retval);
1952:   if (!EG(exception) && Z_TYPE(retval) != IS_UNDEF) {
1953:     spl_instantiate_arg_ex1(Z_OBJCE_P(getThis()), return_value, &retval);
1954:   }
1955:   zval_ptr_dtor(&retval);
1956: } /* }}} */
1957:
1958: /* {{{ proto RecursiveCallbackFilterIterator RecursiveCallbackFilterIterator::getChildren()
1959:    Return the inner iterator's children contained in a RecursiveCallbackFilterIterator */
1960: SPL_METHOD(RecursiveCallbackFilterIterator, getChildren)
1961: {
1962:   spl_dual_it_object   *intern;
1963:   zval                 retval;
1964:
1965:   if (zend_parse_parameters_none() == FAILURE) {
1966:     return;
1967:   }
1968:
1969:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1970:
1971:   zend_call_method_with_0_params(&intern->inner.zobject, intern->inner.ce, NULL, "getchildren", &retval);
1972:   if (!EG(exception) && Z_TYPE(retval) != IS_UNDEF) {
1973:     spl_instantiate_arg_ex2(Z_OBJCE_P(getThis()), return_value, &retval, &intern->u.cbfilter->fci.function_name);
1974:   }
1975:   zval_ptr_dtor(&retval);
1976: } /* }}} */
1977: /* {{{ proto void ParentIterator::__construct(RecursiveIterator it)
1978:    Create a ParentIterator from a RecursiveIterator */
1979: SPL_METHOD(ParentIterator, __construct)
1980: {
1981:   spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_ParentIterator, spl_ce_RecursiveIterator, DIT_ParentIterator);
1982: } /* }}} */
1983:
1984: #if HAVE_PCRE || HAVE_BUNDLED_PCRE
1985: /* {{{ proto void RegexIterator::__construct(Iterator it, string regex [, int mode [, int flags [, int preg_flags]]])
1986:    Create an RegexIterator from another iterator and a regular expression */
1987: SPL_METHOD(RegexIterator, __construct)
1988: {
1989:   spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RegexIterator, zend_ce_iterator, DIT_RegexIterator);
1990: } /* }}} */
1991:
1992: /* {{{ proto bool CallbackFilterIterator::accept()
1993:    Calls the callback with the current value, the current key and the inner iterator as arguments */
1994: SPL_METHOD(CallbackFilterIterator, accept)
1995: {
1996:   spl_dual_it_object     *intern = Z_SPLDUAL_IT_P(getThis());
1997:   zend_fcall_info        *fci = &intern->u.cbfilter->fci;
1998:   zend_fcall_info_cache  *fcc = &intern->u.cbfilter->fcc;
1999:   zval                    params[3];
2000:
2001:   if (zend_parse_parameters_none() == FAILURE) {
2002:     return;
2003:   }
2004:
2005:   if (Z_TYPE(intern->current.data) == IS_UNDEF || Z_TYPE(intern->current.key) == IS_UNDEF) {
2006:     RETURN_FALSE;
2007:   }
2008:
2009:   ZVAL_COPY_VALUE(&params[0], &intern->current.data);
2010:   ZVAL_COPY_VALUE(&params[1], &intern->current.key);
2011:   ZVAL_COPY_VALUE(&params[2], &intern->inner.zobject);
2012:
2013:   fci->retval = return_value;
2014:   fci->param_count = 3;
2015:   fci->params = params;
2016:   fci->no_separation = 0;
2017:
2018:   if (zend_call_function(fci, fcc) != SUCCESS || Z_ISUNDEF_P(return_value)) {
2019:     RETURN_FALSE;
2020:   }
2021:
2022:   if (EG(exception)) {
2023:     RETURN_NULL();
2024:   }
2025:
2026:   /* zend_call_function may change args to IS_REF */
2027:   ZVAL_COPY_VALUE(&intern->current.data, &params[0]);
2028:   ZVAL_COPY_VALUE(&intern->current.key, &params[1]);
2029: }
2030: /* }}} */
2031:
2032: /* {{{ proto bool RegexIterator::accept()
2033:    Match (string)current() against regular expression */
2034: SPL_METHOD(RegexIterator, accept)
2035: {
2036:   spl_dual_it_object *intern;
2037:   zend_string *result, *subject;
2038:   size_t count = 0;
2039:   zval zcount, *replacement, tmp_replacement, rv;
2040:   pcre2_match_data *match_data;
2041:   pcre2_code *re;
2042:   int rc;
2043:
2044:   if (zend_parse_parameters_none() == FAILURE) {
2045:     return;
2046:   }
2047:
2048:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2049:
2050:   if (Z_TYPE(intern->current.data) == IS_UNDEF) {
2051:     RETURN_FALSE;
2052:   }
2053:
2054:   if (intern->u.regex.flags & REGIT_USE_KEY) {
2055:     subject = zval_get_string(&intern->current.key);
2056:   } else {
2057:     if (Z_TYPE(intern->current.data) == IS_ARRAY) {
2058:       RETURN_FALSE;
2059:     }
2060:     subject = zval_get_string(&intern->current.data);
2061:   }
2062:
2063:   switch (intern->u.regex.mode)
2064:   {
2065:     case REGIT_MODE_MAX: /* won't happen but makes compiler happy */
2066:     case REGIT_MODE_MATCH:
2067:       re = php_pcre_pce_re(intern->u.regex.pce);
2068:       match_data = php_pcre_create_match_data(0, re);
2069:       if (!match_data) {
2070:         RETURN_FALSE;
2071:       }
2072:       rc = pcre2_match(re, (PCRE2_SPTR)ZSTR_VAL(subject), ZSTR_LEN(subject), 0, 0, match_data, php_pcre_mctx());
2073:       RETVAL_BOOL(rc >= 0);
2074:       php_pcre_free_match_data(match_data);
2075:       break;
2076:
2077:     case REGIT_MODE_ALL_MATCHES:
2078:     case REGIT_MODE_GET_MATCH:
2079:       zval_ptr_dtor(&intern->current.data);
2080:       ZVAL_UNDEF(&intern->current.data);
2081:       php_pcre_match_impl(intern->u.regex.pce, ZSTR_VAL(subject), ZSTR_LEN(subject), &zcount,
2082:         &intern->current.data, intern->u.regex.mode == REGIT_MODE_ALL_MATCHES, intern->u.regex.use_flags, intern->u.regex.preg_flags, 0);
2083:       RETVAL_BOOL(Z_LVAL(zcount) > 0);
2084:       break;
2085:
2086:     case REGIT_MODE_SPLIT:
2087:       zval_ptr_dtor(&intern->current.data);
2088:       ZVAL_UNDEF(&intern->current.data);
2089:       php_pcre_split_impl(intern->u.regex.pce, subject, &intern->current.data, -1, intern->u.regex.preg_flags);
2090:       count = zend_hash_num_elements(Z_ARRVAL(intern->current.data));
2091:       RETVAL_BOOL(count > 1);
2092:       break;
2093:
2094:     case REGIT_MODE_REPLACE:
2095:       replacement = zend_read_property(intern->std.ce, getThis(), "replacement", sizeof("replacement")-1, 1, &rv);
2096:       if (Z_TYPE_P(replacement) != IS_STRING) {
2097:         ZVAL_COPY(&tmp_replacement, replacement);
2098:         convert_to_string(&tmp_replacement);
2099:         replacement = &tmp_replacement;
2100:       }
2101:       result = php_pcre_replace_impl(intern->u.regex.pce, subject, ZSTR_VAL(subject), ZSTR_LEN(subject), Z_STR_P(replacement), -1, &count);
2102:
2103:       if (intern->u.regex.flags & REGIT_USE_KEY) {
2104:         zval_ptr_dtor(&intern->current.key);
2105:         ZVAL_STR(&intern->current.key, result);
2106:       } else {
2107:         zval_ptr_dtor(&intern->current.data);
2108:         ZVAL_STR(&intern->current.data, result);
2109:       }
2110:
2111:       if (replacement == &tmp_replacement) {
2112:         zval_ptr_dtor(replacement);
2113:       }
2114:       RETVAL_BOOL(count > 0);
2115:   }
2116:
2117:   if (intern->u.regex.flags & REGIT_INVERTED) {
2118:     RETVAL_BOOL(Z_TYPE_P(return_value) != IS_TRUE);
2119:   }
2120:   zend_string_release(subject);
2121: } /* }}} */
2122:
2123: /* {{{ proto string RegexIterator::getRegex()
2124:    Returns current regular expression */
2125: SPL_METHOD(RegexIterator, getRegex)
2126: {
2127:   spl_dual_it_object *intern = Z_SPLDUAL_IT_P(getThis());
2128:
2129:   if (zend_parse_parameters_none() == FAILURE) {
2130:     return;
2131:   }
2132:
2133:   RETURN_STR_COPY(intern->u.regex.regex);
2134: } /* }}} */
2135:
2136: /* {{{ proto bool RegexIterator::getMode()
2137:    Returns current operation mode */
2138: SPL_METHOD(RegexIterator, getMode)
2139: {
2140:   spl_dual_it_object *intern;
2141:
2142:   if (zend_parse_parameters_none() == FAILURE) {
2143:     return;
2144:   }
2145:
2146:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2147:
2148:   RETURN_LONG(intern->u.regex.mode);
2149: } /* }}} */
2150:
2151: /* {{{ proto bool RegexIterator::setMode(int new_mode)
2152:    Set new operation mode */
2153: SPL_METHOD(RegexIterator, setMode)
2154: {
2155:   spl_dual_it_object *intern;
2156:   zend_long mode;
2157:
2158:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &mode) == FAILURE) {
2159:     return;
2160:   }
2161:
2162:   if (mode < 0 || mode >= REGIT_MODE_MAX) {
2163:     zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "Illegal mode " ZEND_LONG_FMT, mode);
2164:     return;/* NULL */
2165:   }
2166:
2167:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2168:
2169:   intern->u.regex.mode = mode;
2170: } /* }}} */
2171:
2172: /* {{{ proto bool RegexIterator::getFlags()
2173:    Returns current operation flags */
2174: SPL_METHOD(RegexIterator, getFlags)
2175: {
2176:   spl_dual_it_object *intern;
2177:
2178:   if (zend_parse_parameters_none() == FAILURE) {
2179:     return;
2180:   }
2181:
2182:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2183:
2184:   RETURN_LONG(intern->u.regex.flags);
2185: } /* }}} */
2186:
2187: /* {{{ proto bool RegexIterator::setFlags(int new_flags)
2188:    Set operation flags */
2189: SPL_METHOD(RegexIterator, setFlags)
2190: {
2191:   spl_dual_it_object *intern;
2192:   zend_long flags;
2193:
2194:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &flags) == FAILURE) {
2195:     return;
2196:   }
2197:
2198:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2199:
2200:   intern->u.regex.flags = flags;
2201: } /* }}} */
2202:
2203: /* {{{ proto bool RegexIterator::getFlags()
2204:    Returns current PREG flags (if in use or NULL) */
2205: SPL_METHOD(RegexIterator, getPregFlags)
2206: {
2207:   spl_dual_it_object *intern;
2208:
2209:   if (zend_parse_parameters_none() == FAILURE) {
2210:     return;
2211:   }
2212:
2213:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2214:
2215:   if (intern->u.regex.use_flags) {
2216:     RETURN_LONG(intern->u.regex.preg_flags);
2217:   } else {
2218:     return;
2219:   }
2220: } /* }}} */
2221:
2222: /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2223:    Set PREG flags */
2224: SPL_METHOD(RegexIterator, setPregFlags)
2225: {
2226:   spl_dual_it_object *intern;
2227:   zend_long preg_flags;
2228:
2229:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &preg_flags) == FAILURE) {
2230:     return;
2231:   }
2232:
2233:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2234:
2235:   intern->u.regex.preg_flags = preg_flags;
2236:   intern->u.regex.use_flags = 1;
2237: } /* }}} */
2238:
2239: /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2240:    Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2241: SPL_METHOD(RecursiveRegexIterator, __construct)
2242: {
2243:   spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveRegexIterator, spl_ce_RecursiveIterator, DIT_RecursiveRegexIterator);
2244: } /* }}} */
2245:
```

```
2246: /* {{{ proto RecursiveRegexIterator RecursiveRegexIterator::getChildren()
2247:    Return the inner iterator's children contained in a RecursiveRegexIterator */
2248: SPL_METHOD(RecursiveRegexIterator, getChildren)
2249: {
2250:   spl_dual_it_object   *intern;
2251:   zval                 retval;
2252:
2253:   if (zend_parse_parameters_none() == FAILURE) {
2254:     return;
2255:   }
2256:
2257:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2258:
2259:   zend_call_method_with_0_params(&intern->inner.zobject, intern->inner.ce, NULL, "getchildren", &retval);
2260:   if (!EG(exception)) {
2261:     zval args[5];
2262:
2263:     ZVAL_COPY(&args[0], &retval);
2264:     ZVAL_STR_COPY(&args[1], intern->u.regex.regex);
2265:     ZVAL_LONG(&args[2], intern->u.regex.mode);
2266:     ZVAL_LONG(&args[3], intern->u.regex.flags);
2267:     ZVAL_LONG(&args[4], intern->u.regex.preg_flags);
2268:
2269:     spl_instantiate_arg_n(Z_OBJCE_P(getThis()), return_value, 5, args);
2270:
2271:     zval_ptr_dtor(&args[0]);
2272:     zval_ptr_dtor(&args[1]);
2273:   }
2274:   zval_ptr_dtor(&retval);
2275: } /* }}} */
2276:
2277: SPL_METHOD(RecursiveRegexIterator, accept)
2278: {
2279:   spl_dual_it_object *intern;
2280:
2281:   if (zend_parse_parameters_none() == FAILURE) {
2282:     return;
2283:   }
2284:
2285:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2286:
2287:   if (Z_TYPE(intern->current.data) == IS_UNDEF) {
2288:     RETURN_FALSE;
2289:   } else if (Z_TYPE(intern->current.data) == IS_ARRAY) {
2290:     RETURN_BOOL(zend_hash_num_elements(Z_ARRVAL(intern->current.data)) > 0);
2291:   }
2292:
2293:   zend_call_method_with_0_params(getThis(), spl_ce_RegexIterator, NULL, "accept", return_value);
2294: }
2295:
2296: #endif
2297:
2298: /* {{{ spl_dual_it_dtor */
2299: static void spl_dual_it_dtor(zend_object *_object)
2300: {
2301:   spl_dual_it_object *object = spl_dual_it_from_obj(_object);
2302:
2303:   /* call standard dtor */
2304:   zend_objects_destroy_object(_object);
2305:
2306:   spl_dual_it_free(object);
2307:
2308:   if (object->inner.iterator) {
2309:     zend_iterator_dtor(object->inner.iterator);
2310:   }
2311: }
2312: /* }}} */
2313:
2314: /* {{{ spl_dual_it_free_storage */
2315: static void spl_dual_it_free_storage(zend_object *_object)
2316: {
2317:   spl_dual_it_object *object = spl_dual_it_from_obj(_object);
2318:
2319:
2320:   if (!Z_ISUNDEF(object->inner.zobject)) {
2321:     zval_ptr_dtor(&object->inner.zobject);
2322:   }
2323:
2324:   if (object->dit_type == DIT_AppendIterator) {
2325:     zend_iterator_dtor(object->u.append.iterator);
2326:     if (Z_TYPE(object->u.append.zarrayit) != IS_UNDEF) {
2327:       zval_ptr_dtor(&object->u.append.zarrayit);
2328:     }
2329:   }
2330:
2331:   if (object->dit_type == DIT_CachingIterator || object->dit_type == DIT_RecursiveCachingIterator) {
2332:     zval_ptr_dtor(&object->u.caching.zcache);
2333:   }
2334:
2335: #if HAVE_PCRE || HAVE_BUNDLED_PCRE
2336:   if (object->dit_type == DIT_RegexIterator || object->dit_type == DIT_RecursiveRegexIterator) {
2337:     if (object->u.regex.pce) {
2338:       php_pcre_pce_decref(object->u.regex.pce);
2339:     }
2340:     if (object->u.regex.regex) {
2341:       zend_string_release(object->u.regex.regex);
2342:     }
2343:   }
2344: #endif
2345:
2346:   if (object->dit_type == DIT_CallbackFilterIterator || object->dit_type == DIT_RecursiveCallbackFilterIterator) {
2347:     if (object->u.cbfilter) {
2348:       _spl_cbfilter_it_intern *cbfilter = object->u.cbfilter;
2349:       object->u.cbfilter = NULL;
2350:       zval_ptr_dtor(&cbfilter->fci.function_name);
2351:       if (cbfilter->fci.object) {
2352:         OBJ_RELEASE(cbfilter->fci.object);
2353:       }
2354:       efree(cbfilter);
2355:     }
2356:   }
2357:
2358:   zend_object_std_dtor(&object->std);
2359: }
2360: /* }}} */
2361:
2362: /* {{{ spl_dual_it_new */
2363: static zend_object *spl_dual_it_new(zend_class_entry *class_type)
2364: {
2365:   spl_dual_it_object *intern;
2366:
2367:   intern = zend_object_alloc(sizeof(spl_dual_it_object), class_type);
2368:   intern->dit_type = DIT_Unknown;
2369:
2370:   zend_object_std_init(&intern->std, class_type);
2371:   object_properties_init(&intern->std, class_type);
2372:
2373:   intern->std.handlers = &spl_handlers_dual_it;
2374:   return &intern->std;
2375: }
2376: /* }}} */
2377:
2378: ZEND_BEGIN_ARG_INFO(arginfo_filter_it___construct, 0)
2379:   ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
2380: ZEND_END_ARG_INFO();
2381:
2382: static const zend_function_entry spl_funcs_FilterIterator[] = {
2383:   SPL_ME(FilterIterator,  __construct,      arginfo_filter_it___construct, ZEND_ACC_PUBLIC)
2384:   SPL_ME(FilterIterator,  rewind,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2385:   SPL_ME(dual_it,         valid,            arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2386:   SPL_ME(dual_it,         key,              arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2387:   SPL_ME(dual_it,         current,          arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2388:   SPL_ME(FilterIterator,  next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2389:   SPL_ME(dual_it,         getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2390:   SPL_ABSTRACT_ME(FilterIterator, accept,   arginfo_recursive_it_void)
2391:   PHP_FE_END
2392: };
2393:
2394: ZEND_BEGIN_ARG_INFO(arginfo_callback_filter_it___construct, 0)
2395:   ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
2396:   ZEND_ARG_INFO(0, callback)
2397: ZEND_END_ARG_INFO();
2398:
2399: static const zend_function_entry spl_funcs_CallbackFilterIterator[] = {
2400:   SPL_ME(CallbackFilterIterator, __construct, arginfo_callback_filter_it___construct, ZEND_ACC_PUBLIC)
2401:   SPL_ME(CallbackFilterIterator, accept,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2402:   PHP_FE_END
2403: };
2404:
2405: ZEND_BEGIN_ARG_INFO(arginfo_recursive_callback_filter_it___construct, 0)
2406:   ZEND_ARG_OBJ_INFO(0, iterator, RecursiveIterator, 0)
2407:   ZEND_ARG_INFO(0, callback)
2408: ZEND_END_ARG_INFO();
2409:
2410: static const zend_function_entry spl_funcs_RecursiveCallbackFilterIterator[] = {
2411:   SPL_ME(RecursiveCallbackFilterIterator, __construct, arginfo_recursive_callback_filter_it___construct, ZEND_ACC_PUBLIC)
2412:   SPL_ME(RecursiveFilterIterator,  hasChildren,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2413:   SPL_ME(RecursiveCallbackFilterIterator,  getChildren,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2414:   PHP_FE_END
2415: };
2416:
2417: ZEND_BEGIN_ARG_INFO(arginfo_parent_it___construct, 0)
2418:   ZEND_ARG_OBJ_INFO(0, iterator, RecursiveIterator, 0)
2419: ZEND_END_ARG_INFO();
2420:
2421: static const zend_function_entry spl_funcs_RecursiveFilterIterator[] = {
2422:   SPL_ME(RecursiveFilterIterator,  __construct,      arginfo_parent_it___construct, ZEND_ACC_PUBLIC)
2423:   SPL_ME(RecursiveFilterIterator,  hasChildren,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2424:   SPL_ME(RecursiveFilterIterator,  getChildren,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2425:   PHP_FE_END
2426: };
2427:
2428: static const zend_function_entry spl_funcs_ParentIterator[] = {
2429:   SPL_ME(ParentIterator,  __construct,      arginfo_parent_it___construct, ZEND_ACC_PUBLIC)
2430:   SPL_MA(ParentIterator,  accept,           RecursiveFilterIterator, hasChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2431:   PHP_FE_END
2432: };
2433:
2434: #if HAVE_PCRE || HAVE_BUNDLED_PCRE
2435: ZEND_BEGIN_ARG_INFO_EX(arginfo_regex_it___construct, 0, 0, 2)
2436:   ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
2437:   ZEND_ARG_INFO(0, regex)
2438:   ZEND_ARG_INFO(0, mode)
2439:   ZEND_ARG_INFO(0, flags)
2440:   ZEND_ARG_INFO(0, preg_flags)
2441: ZEND_END_ARG_INFO();
2442:
2443: ZEND_BEGIN_ARG_INFO_EX(arginfo_regex_it_set_mode, 0, 0, 1)
2444:   ZEND_ARG_INFO(0, mode)
2445: ZEND_END_ARG_INFO();
2446:
2447: ZEND_BEGIN_ARG_INFO_EX(arginfo_regex_it_set_flags, 0, 0, 1)
2448:   ZEND_ARG_INFO(0, flags)
2449: ZEND_END_ARG_INFO();
2450:
2451: ZEND_BEGIN_ARG_INFO_EX(arginfo_regex_it_set_preg_flags, 0, 0, 1)
2452:   ZEND_ARG_INFO(0, preg_flags)
2453: ZEND_END_ARG_INFO();
2454:
2455: static const zend_function_entry spl_funcs_RegexIterator[] = {
2456:   SPL_ME(RegexIterator,   __construct,      arginfo_regex_it___construct,    ZEND_ACC_PUBLIC)
2457:   SPL_ME(RegexIterator,   accept,           arginfo_recursive_it_void,       ZEND_ACC_PUBLIC)
2458:   SPL_ME(RegexIterator,   getMode,          arginfo_recursive_it_void,       ZEND_ACC_PUBLIC)
2459:   SPL_ME(RegexIterator,   setMode,          arginfo_regex_it_set_mode,       ZEND_ACC_PUBLIC)
2460:   SPL_ME(RegexIterator,   getFlags,         arginfo_recursive_it_void,       ZEND_ACC_PUBLIC)
2461:   SPL_ME(RegexIterator,   setFlags,         arginfo_regex_it_set_flags,      ZEND_ACC_PUBLIC)
2462:   SPL_ME(RegexIterator,   getPregFlags,     arginfo_recursive_it_void,       ZEND_ACC_PUBLIC)
2463:   SPL_ME(RegexIterator,   setPregFlags,     arginfo_regex_it_set_preg_flags, ZEND_ACC_PUBLIC)
2464:   SPL_ME(RegexIterator,   getRegex,         arginfo_recursive_it_void,       ZEND_ACC_PUBLIC)
2465:   PHP_FE_END
2466: };
2467:
2468: ZEND_BEGIN_ARG_INFO_EX(arginfo_rec_regex_it___construct, 0, 0, 2)
2469:   ZEND_ARG_OBJ_INFO(0, iterator, RecursiveIterator, 0)
2470:   ZEND_ARG_INFO(0, regex)
2471:   ZEND_ARG_INFO(0, mode)
2472:   ZEND_ARG_INFO(0, flags)
2473:   ZEND_ARG_INFO(0, preg_flags)
2474: ZEND_END_ARG_INFO();
2475:
2476: static const zend_function_entry spl_funcs_RecursiveRegexIterator[] = {
2477:   SPL_ME(RecursiveRegexIterator,  __construct,      arginfo_rec_regex_it___construct, ZEND_ACC_PUBLIC)
2478:   SPL_ME(RecursiveRegexIterator,  accept,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2479:   SPL_ME(RecursiveFilterIterator, hasChildren,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2480:   SPL_ME(RecursiveRegexIterator,  getChildren,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2481:   PHP_FE_END
2482: };
2483: #endif
2484:
2485: static inline int spl_limit_it_valid(spl_dual_it_object *intern)
2486: {
2487:   /* FAILURE / SUCCESS */
2488:   if (intern->u.limit.count != -1 && intern->current.pos >= intern->u.limit.offset + intern->u.limit.count) {
2489:     return FAILURE;
2490:   } else {
2491:     return spl_dual_it_valid(intern);
2492:   }
2493: }
2494:
2495: static inline void spl_limit_it_seek(spl_dual_it_object *intern, zend_long pos)
2496: {
2497:   zval  zpos;
2498:
2499:   spl_dual_it_free(intern);
2500:   if (pos < intern->u.limit.offset) {
2501:     zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Cannot seek to " ZEND_LONG_FMT " which is below the offset " ZEND_LONG_FMT, pos, intern->u
.limit.offset);
2502:     return;
2503:   }
2504:   if (pos >= intern->u.limit.offset + intern->u.limit.count && intern->u.limit.count != -1) {
2505:     zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Cannot seek to " ZEND_LONG_FMT " which is behind offset " ZEND_LONG_FMT " plus count " ZEN
D_LONG_FMT, pos, intern->u.limit.offset, intern->u.limit.count);
2506:     return;
2507:   }
2508:   if (pos != intern->current.pos && instanceof_function(intern->inner.ce, spl_ce_SeekableIterator)) {
2509:     ZVAL_LONG(&zpos, pos);
2510:     spl_dual_it_free(intern);
2511:     zend_call_method_with_1_params(&intern->inner.zobject, intern->inner.ce, NULL, "seek", NULL, &zpos);
2512:     zval_ptr_dtor(&zpos);
2513:     if (!EG(exception)) {
2514:       intern->current.pos = pos;
2515:       if (spl_limit_it_valid(intern) == SUCCESS) {
2516:         spl_dual_it_fetch(intern, 0);
2517:       }
2518:     }
2519:   } else {
2520:     /* emulate the forward seek, by next() calls */
2521:     /* a back ward seek is done by a previous rewind() */
2522:     if (pos < intern->current.pos) {
2523:       spl_dual_it_rewind(intern);
2524:     }
2525:     while (pos > intern->current.pos && spl_dual_it_valid(intern) == SUCCESS) {
2526:       spl_dual_it_next(intern, 1);
2527:     }
2528:     if (spl_dual_it_valid(intern) == SUCCESS) {
2529:       spl_dual_it_fetch(intern, 1);
2530:     }
2531:   }
2532: }
2533:
2534: /* {{{ proto void LimitIterator::__construct(Iterator it [, int offset, int count])
2535:    Construct a LimitIterator from an Iterator with a given starting offset and optionally a maximum count */
2536: SPL_METHOD(LimitIterator, __construct)
2537: {
2538:   spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_LimitIterator, zend_ce_iterator, DIT_LimitIterator);
2539: } /* }}} */
2540:
2541: /* {{{ proto void LimitIterator::rewind()
2542:    Rewind the iterator to the specified starting offset */
2543: SPL_METHOD(LimitIterator, rewind)
2544: {
2545:   spl_dual_it_object   *intern;
2546:
2547:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2548:   spl_dual_it_rewind(intern);
2549:   spl_limit_it_seek(intern, intern->u.limit.offset);
2550: } /* }}} */
2551:
2552: /* {{{ proto bool LimitIterator::valid()
2553:    Check whether the current element is valid */
2554: SPL_METHOD(LimitIterator, valid)
2555: {
2556:   spl_dual_it_object   *intern;
2557:
2558:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2559:
2560:   /*  RETURN_BOOL(spl_limit_it_valid(intern) == SUCCESS);*/
2561:   RETURN_BOOL((intern->u.limit.count == -1 || intern->current.pos < intern->u.limit.offset + intern->u.limit.count) && Z_TYPE(intern->current.data) !=
IS_UNDEF);
2562: } /* }}} */
2563:
2564: /* {{{ proto void LimitIterator::next()
2565:    Move the iterator forward */
2566: SPL_METHOD(LimitIterator, next)
2567: {
2568:   spl_dual_it_object   *intern;
2569:
2570:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2571:
2572:   spl_dual_it_next(intern, 1);
2573:   if (intern->u.limit.count == -1 || intern->current.pos < intern->u.limit.offset + intern->u.limit.count) {
2574:     spl_dual_it_fetch(intern, 1);
2575:   }
2576: } /* }}} */
2577:
2578: /* {{{ proto void LimitIterator::seek(int position)
2579:    Seek to the given position */
2580: SPL_METHOD(LimitIterator, seek)
2581: {
2582:   spl_dual_it_object   *intern;
2583:   zend_long             pos;
2584:
2585:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &pos) == FAILURE) {
2586:     return;
2587:   }
2588:
2589:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2590:   spl_limit_it_seek(intern, pos);
2591:   RETURN_LONG(intern->current.pos);
2592: } /* }}} */
2593:
2594: /* {{{ proto int LimitIterator::getPosition()
2595:    Return the current position */
2596: SPL_METHOD(LimitIterator, getPosition)
2597: {
2598:   spl_dual_it_object   *intern;
2599:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2600:   RETURN_LONG(intern->current.pos);
2601: } /* }}} */
2602:
2603: ZEND_BEGIN_ARG_INFO(arginfo_seekable_it_seek, 0)
2604:   ZEND_ARG_INFO(0, position)
2605: ZEND_END_ARG_INFO();
2606:
2607: static const zend_function_entry spl_funcs_SeekableIterator[] = {
2608:   SPL_ABSTRACT_ME(SeekableIterator, seek, arginfo_seekable_it_seek)
2609:   PHP_FE_END
2610: };
2611:
2612: ZEND_BEGIN_ARG_INFO_EX(arginfo_limit_it___construct, 0, 0, 1)
2613:   ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
2614:   ZEND_ARG_INFO(0, offset)
2615:   ZEND_ARG_INFO(0, count)
2616: ZEND_END_ARG_INFO();
2617:
2618: ZEND_BEGIN_ARG_INFO(arginfo_limit_it_seek, 0)
```

```
2619:   ZEND_ARG_INFO(0, position)
2620: ZEND_END_ARG_INFO();
2621:
2622: static const zend_function_entry spl_funcs_LimitIterator[] = {
2623:   SPL_ME(LimitIterator,   __construct,       arginfo_limit_it___construct, ZEND_ACC_PUBLIC)
2624:   SPL_ME(LimitIterator,   rewind,            arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2625:   SPL_ME(LimitIterator,   valid,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2626:   SPL_ME(dual_it,         key,               arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2627:   SPL_ME(dual_it,         current,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2628:   SPL_ME(LimitIterator,   next,              arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2629:   SPL_ME(LimitIterator,   seek,              arginfo_limit_it_seek, ZEND_ACC_PUBLIC)
2630:   SPL_ME(LimitIterator,   getPosition,       arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2631:   SPL_ME(dual_it,         getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2632:   PHP_FE_END
2633: };
2634:
2635: static inline int spl_caching_it_valid(spl_dual_it_object *intern)
2636: {
2637:   return intern->u.caching.flags & CIT_VALID ? SUCCESS : FAILURE;
2638: }
2639:
2640: static inline int spl_caching_it_has_next(spl_dual_it_object *intern)
2641: {
2642:   return spl_dual_it_valid(intern);
2643: }
2644:
2645: static inline void spl_caching_it_next(spl_dual_it_object *intern)
2646: {
2647:   if (spl_dual_it_fetch(intern, 1) == SUCCESS) {
2648:     intern->u.caching.flags |= CIT_VALID;
2649:     /* Full cache ? */
2650:     if (intern->u.caching.flags & CIT_FULL_CACHE) {
2651:       zval *key = &intern->current.key;
2652:       zval *data = &intern->current.data;
2653:
2654:       ZVAL_DEREF(data);
2655:       Z_TRY_ADDREF_P(data);
2656:       array_set_zval_key(Z_ARRVAL(intern->u.caching.zcache), key, data);
2657:       zval_ptr_dtor(data);
2658:     }
2659:     /* Recursion ? */
2660:     if (intern->dit_type == DIT_RecursiveCachingIterator) {
2661:       zval retval, zchildren, zflags;
2662:       zend_call_method_with_0_params(&intern->inner.zobject, intern->inner.ce, NULL, "haschildren", &retval);
2663:       if (EG(exception)) {
2664:         zval_ptr_dtor(&retval);
2665:         if (intern->u.caching.flags & CIT_CATCH_GET_CHILD) {
2666:           zend_clear_exception();
2667:         } else {
2668:           return;
2669:         }
2670:       } else {
2671:         if (zend_is_true(&retval)) {
2672:           zend_call_method_with_0_params(&intern->inner.zobject, intern->inner.ce, NULL, "getchildren", &zchildren);
2673:           if (EG(exception)) {
2674:             zval_ptr_dtor(&zchildren);
2675:             if (intern->u.caching.flags & CIT_CATCH_GET_CHILD) {
2676:               zend_clear_exception();
2677:             } else {
2678:               zval_ptr_dtor(&retval);
2679:               return;
2680:             }
2681:           } else {
2682:             ZVAL_LONG(&zflags, intern->u.caching.flags & CIT_PUBLIC);
2683:             spl_instantiate_arg_ex2(spl_ce_RecursiveCachingIterator, &intern->u.caching.zchildren, &zchildren, &zflags);
2684:             zval_ptr_dtor(&zchildren);
2685:           }
2686:         }
2687:         zval_ptr_dtor(&retval);
2688:         if (EG(exception)) {
2689:           if (intern->u.caching.flags & CIT_CATCH_GET_CHILD) {
2690:             zend_clear_exception();
2691:           } else {
2692:             return;
2693:           }
2694:         }
2695:       }
2696:     }
2697:     if (intern->u.caching.flags & (CIT_TOSTRING_USE_INNER|CIT_CALL_TOSTRING)) {
2698:       int use_copy;
2699:       zval expr_copy;
2700:       if (intern->u.caching.flags & CIT_TOSTRING_USE_INNER) {
2701:         ZVAL_COPY_VALUE(&intern->u.caching.zstr, &intern->inner.zobject);
2702:       } else {
2703:         ZVAL_COPY_VALUE(&intern->u.caching.zstr, &intern->current.data);
2704:       }
2705:       use_copy = zend_make_printable_zval(&intern->u.caching.zstr, &expr_copy);
2706:       if (use_copy) {
2707:         ZVAL_COPY_VALUE(&intern->u.caching.zstr, &expr_copy);
2708:       } else {
2709:         Z_TRY_ADDREF(intern->u.caching.zstr);
2710:       }
2711:     }
2712:     spl_dual_it_next(intern, 0);
2713:   } else {
2714:     intern->u.caching.flags &= ~CIT_VALID;
2715:   }
2716: }
2717:
2718: static inline void spl_caching_it_rewind(spl_dual_it_object *intern)
2719: {
2720:   spl_dual_it_rewind(intern);
2721:   zend_hash_clean(Z_ARRVAL(intern->u.caching.zcache));
2722:   spl_caching_it_next(intern);
2723: }
2724:
2725: /* {{{ proto void CachingIterator::__construct(Iterator it [, flags = CIT_CALL_TOSTRING])
2726:    Construct a CachingIterator from an Iterator */
2727: SPL_METHOD(CachingIterator, __construct)
2728: {
2729:   spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_CachingIterator, zend_ce_iterator, DIT_CachingIterator);
2730: } /* }}} */
2731:
2732: /* {{{ proto void CachingIterator::rewind()
2733:    Rewind the iterator */
2734: SPL_METHOD(CachingIterator, rewind)
2735: {
2736:   spl_dual_it_object   *intern;
2737:
2738:   if (zend_parse_parameters_none() == FAILURE) {
2739:     return;
2740:   }
2741:
2742:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2743:
2744:   spl_caching_it_rewind(intern);
2745: } /* }}} */
2746:
2747: /* {{{ proto bool CachingIterator::valid()
2748:    Check whether the current element is valid */
2749: SPL_METHOD(CachingIterator, valid)
2750: {
2751:   spl_dual_it_object   *intern;
2752:
2753:   if (zend_parse_parameters_none() == FAILURE) {
2754:     return;
2755:   }
2756:
2757:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2758:
2759:   RETURN_BOOL(spl_caching_it_valid(intern) == SUCCESS);
2760: } /* }}} */
2761:
2762: /* {{{ proto void CachingIterator::next()
2763:    Move the iterator forward */
2764: SPL_METHOD(CachingIterator, next)
2765: {
2766:   spl_dual_it_object   *intern;
2767:
2768:   if (zend_parse_parameters_none() == FAILURE) {
2769:     return;
2770:   }
2771:
2772:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2773:
2774:   spl_caching_it_next(intern);
2775: } /* }}} */
2776:
2777: /* {{{ proto bool CachingIterator::hasNext()
2778:    Check whether the inner iterator has a valid next element */
2779: SPL_METHOD(CachingIterator, hasNext)
2780: {
2781:   spl_dual_it_object   *intern;
2782:
2783:   if (zend_parse_parameters_none() == FAILURE) {
2784:     return;
2785:   }
2786:
2787:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2788:
2789:   RETURN_BOOL(spl_caching_it_has_next(intern) == SUCCESS);
2790: } /* }}} */
2791:
2792: /* {{{ proto string CachingIterator::__toString()
2793:    Return the string representation of the current element */
2794: SPL_METHOD(CachingIterator, __toString)
2795: {
2796:   spl_dual_it_object   *intern;
2797:
2798:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2799:
2800:   if (!(intern->u.caching.flags & (CIT_CALL_TOSTRING|CIT_TOSTRING_USE_KEY|CIT_TOSTRING_USE_CURRENT|CIT_TOSTRING_USE_INNER))) {
2801:     zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not fetch string value (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis())->name));
2802:     return;
2803:   }
2804:   if (intern->u.caching.flags & CIT_TOSTRING_USE_KEY) {
2805:     ZVAL_COPY(return_value, &intern->current.key);
2806:     convert_to_string(return_value);
2807:     return;
2808:   } else if (intern->u.caching.flags & CIT_TOSTRING_USE_CURRENT) {
2809:     ZVAL_COPY(return_value, &intern->current.data);
2810:     convert_to_string(return_value);
2811:     return;
2812:   }
2813:   if (Z_TYPE(intern->u.caching.zstr) == IS_STRING) {
2814:     RETURN_STR_COPY(Z_STR_P(intern->u.caching.zstr));
2815:   } else {
2816:     RETURN_EMPTY_STRING();
2817:   }
2818: } /* }}} */
2819:
2820: /* {{{ proto void CachingIterator::offsetSet(mixed index, mixed newval)
2821:    Set given index in cache */
2822: SPL_METHOD(CachingIterator, offsetSet)
2823: {
2824:   spl_dual_it_object   *intern;
2825:   zend_string *key;
2826:   zval *value;
2827:
2828:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2829:
2830:   if (!(intern->u.caching.flags & CIT_FULL_CACHE)) {
2831:     zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis())->name));
2832:     return;
2833:   }
2834:
2835:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "Sz", &key, &value) == FAILURE) {
2836:     return;
2837:   }
2838:
2839:   Z_TRY_ADDREF_P(value);
2840:   zend_symtable_update(Z_ARRVAL(intern->u.caching.zcache), key, value);
2841: }
2842: /* }}} */
2843:
2844: /* {{{ proto string CachingIterator::offsetGet(mixed index)
2845:    Return the internal cache if used */
2846: SPL_METHOD(CachingIterator, offsetGet)
2847: {
2848:   spl_dual_it_object   *intern;
2849:   zend_string *key;
2850:   zval *value;
2851:
2852:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2853:
2854:   if (!(intern->u.caching.flags & CIT_FULL_CACHE)) {
2855:     zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis())->name));
2856:     return;
2857:   }
2858:
2859:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "S", &key) == FAILURE) {
2860:     return;
2861:   }
2862:
2863:   if ((value = zend_symtable_find(Z_ARRVAL(intern->u.caching.zcache), key)) == NULL) {
2864:     zend_error(E_NOTICE, "Undefined index: %s", ZSTR_VAL(key));
2865:     return;
2866:   }
2867:
2868:   ZVAL_DEREF(value);
2869:   ZVAL_COPY(return_value, value);
2870: }
2871: /* }}} */
2872:
2873: /* {{{ proto void CachingIterator::offsetUnset(mixed index)
2874:    Unset given index in cache */
2875: SPL_METHOD(CachingIterator, offsetUnset)
2876: {
2877:   spl_dual_it_object   *intern;
2878:   zend_string *key;
2879:
2880:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2881:
2882:   if (!(intern->u.caching.flags & CIT_FULL_CACHE)) {
2883:     zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis())->name));
2884:     return;
2885:   }
2886:
2887:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "S", &key) == FAILURE) {
2888:     return;
2889:   }
2890:
2891:   zend_symtable_del(Z_ARRVAL(intern->u.caching.zcache), key);
2892: }
2893: /* }}} */
2894:
2895: /* {{{ proto bool CachingIterator::offsetExists(mixed index)
2896:    Return whether the requested index exists */
2897: SPL_METHOD(CachingIterator, offsetExists)
2898: {
2899:   spl_dual_it_object   *intern;
2900:   zend_string *key;
2901:
2902:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2903:
2904:   if (!(intern->u.caching.flags & CIT_FULL_CACHE)) {
2905:     zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis())->name));
2906:     return;
2907:   }
2908:
2909:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "S", &key) == FAILURE) {
2910:     return;
2911:   }
2912:
2913:   RETURN_BOOL(zend_symtable_exists(Z_ARRVAL(intern->u.caching.zcache), key));
2914: }
2915: /* }}} */
2916:
2917: /* {{{ proto bool CachingIterator::getCache()
2918:    Return the cache */
2919: SPL_METHOD(CachingIterator, getCache)
2920: {
2921:   spl_dual_it_object   *intern;
2922:
2923:   if (zend_parse_parameters_none() == FAILURE) {
2924:     return;
2925:   }
2926:
2927:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2928:
2929:   if (!(intern->u.caching.flags & CIT_FULL_CACHE)) {
2930:     zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(getThis())->name));
2931:     return;
2932:   }
2933:
2934:   ZVAL_COPY(return_value, &intern->u.caching.zcache);
2935: }
2936: /* }}} */
2937:
2938: /* {{{ proto int CachingIterator::getFlags()
2939:    Return the internal flags */
2940: SPL_METHOD(CachingIterator, getFlags)
2941: {
2942:   spl_dual_it_object   *intern;
2943:
2944:   if (zend_parse_parameters_none() == FAILURE) {
2945:     return;
2946:   }
2947:
2948:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2949:
2950:   RETURN_LONG(intern->u.caching.flags);
2951: }
2952: /* }}} */
2953:
2954: /* {{{ proto void CachingIterator::setFlags(int flags)
2955:    Set the internal flags */
2956: SPL_METHOD(CachingIterator, setFlags)
2957: {
2958:   spl_dual_it_object   *intern;
2959:   zend_long flags;
2960:
2961:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2962:
2963:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &flags) == FAILURE) {
2964:     return;
2965:   }
2966:
2967:   if (spl_cit_check_flags(flags) != SUCCESS) {
2968:     zend_throw_exception(spl_ce_InvalidArgumentException , "Flags must contain only one of CALL_TOSTRING, TOSTRING_USE_KEY, TOSTRING_USE_CURRENT, TOSTRING_USE_INNER", 0);
2969:     return;
2970:   }
2971:   if ((intern->u.caching.flags & CIT_CALL_TOSTRING) != 0 && (flags & CIT_CALL_TOSTRING) == 0) {
2972:     zend_throw_exception(spl_ce_InvalidArgumentException, "Unsetting flag CALL_TO_STRING is not possible", 0);
2973:     return;
2974:   }
2975:   if ((intern->u.caching.flags & CIT_TOSTRING_USE_INNER) != 0 && (flags & CIT_TOSTRING_USE_INNER) == 0) {
2976:     zend_throw_exception(spl_ce_InvalidArgumentException, "Unsetting flag TOSTRING_USE_INNER is not possible", 0);
2977:     return;
2978:   }
2979:   if ((flags & CIT_FULL_CACHE) != 0 && (intern->u.caching.flags & CIT_FULL_CACHE) == 0) {
2980:     /* clear on (re)enable */
2981:     zend_hash_clean(Z_ARRVAL(intern->u.caching.zcache));
2982:   }
2983:   intern->u.caching.flags = (intern->u.caching.flags & ~CIT_PUBLIC) | (flags & CIT_PUBLIC);
2984: }
2985: /* }}} */
2986:
2987: /* {{{ proto void CachingIterator::count()
```

```
2988:    Number of cached elements */
2989: SPL_METHOD(CachingIterator, count)
2990: {
2991:   spl_dual_it_object   *intern;
2992:
2993:   if (zend_parse_parameters_none() == FAILURE) {
2994:     return;
2995:   }
2996:
2997:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2998:
2999:   if (!(intern->u.caching.flags & CIT_FULL_CACHE))  {
3000:     zend_throw_exception_ex(spl_ce_BadMethodCallException, 0, "%s does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_OBJCE_P(get
This())->name));
3001:     return;
3002:   }
3003:
3004:   RETURN_LONG(zend_hash_num_elements(Z_ARRVAL(intern->u.caching.zcache)));
3005: }
3006: /* }}} */
3007:
3008: ZEND_BEGIN_ARG_INFO_EX(arginfo_caching_it___construct, 0, 0, 1)
3009:   ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
3010:   ZEND_ARG_INFO(0, flags)
3011: ZEND_END_ARG_INFO();
3012:
3013: ZEND_BEGIN_ARG_INFO(arginfo_caching_it_setFlags, 0)
3014:   ZEND_ARG_INFO(0, flags)
3015: ZEND_END_ARG_INFO();
3016:
3017: ZEND_BEGIN_ARG_INFO(arginfo_caching_it_offsetGet, 0)
3018:   ZEND_ARG_INFO(0, index)
3019: ZEND_END_ARG_INFO();
3020:
3021: ZEND_BEGIN_ARG_INFO(arginfo_caching_it_offsetSet, 0)
3022:   ZEND_ARG_INFO(0, index)
3023:   ZEND_ARG_INFO(0, newval)
3024: ZEND_END_ARG_INFO();
3025:
3026: static const zend_function_entry spl_funcs_CachingIterator[] = {
3027:   SPL_ME(CachingIterator, __construct,      arginfo_caching_it___construct, ZEND_ACC_PUBLIC)
3028:   SPL_ME(CachingIterator, rewind,           arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
3029:   SPL_ME(CachingIterator, valid,            arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
3030:   SPL_ME(dual_it,         key,              arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
3031:   SPL_ME(dual_it,         current,          arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
3032:   SPL_ME(CachingIterator, next,             arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
3033:   SPL_ME(CachingIterator, hasNext,          arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
3034:   SPL_ME(CachingIterator, __toString,       arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
3035:   SPL_ME(dual_it,         getInnerIterator, arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
3036:   SPL_ME(CachingIterator, getFlags,         arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
3037:   SPL_ME(CachingIterator, setFlags,         arginfo_caching_it_setFlags,    ZEND_ACC_PUBLIC)
3038:   SPL_ME(CachingIterator, offsetGet,        arginfo_caching_it_offsetGet,   ZEND_ACC_PUBLIC)
3039:   SPL_ME(CachingIterator, offsetSet,        arginfo_caching_it_offsetSet,   ZEND_ACC_PUBLIC)
3040:   SPL_ME(CachingIterator, offsetUnset,      arginfo_caching_it_offsetGet,   ZEND_ACC_PUBLIC)
3041:   SPL_ME(CachingIterator, offsetExists,     arginfo_caching_it_offsetGet,   ZEND_ACC_PUBLIC)
3042:   SPL_ME(CachingIterator, getCache,         arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
3043:   SPL_ME(CachingIterator, count,            arginfo_recursive_it_void,      ZEND_ACC_PUBLIC)
3044:   PHP_FE_END
3045: };
3046:
3047: /* {{{ proto void RecursiveCachingIterator::__construct(RecursiveIterator it [, flags = CIT_CALL_TOSTRING])
3048:    Create an iterator from a RecursiveIterator */
3049: SPL_METHOD(RecursiveCachingIterator, __construct)
3050: {
3051:   spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveCachingIterator, spl_ce_RecursiveIterator, DIT_RecursiveCachingIterator);
3052: } /* }}} */
3053:
3054: /* {{{ proto bool RecursiveCachingIterator::hasChildren()
3055:    Check whether the current element of the inner iterator has children */
3056: SPL_METHOD(RecursiveCachingIterator, hasChildren)
3057: {
3058:   spl_dual_it_object   *intern;
3059:
3060:   if (zend_parse_parameters_none() == FAILURE) {
3061:     return;
3062:   }
3063:
3064:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3065:
3066:   RETURN_BOOL(Z_TYPE(intern->u.caching.zchildren) != IS_UNDEF);
3067: } /* }}} */
3068:
3069: /* {{{ proto RecursiveCachingIterator RecursiveCachingIterator::getChildren()
3070:    Return the inner iterator's children as a RecursiveCachingIterator */
3071: SPL_METHOD(RecursiveCachingIterator, getChildren)
3072: {
3073:   spl_dual_it_object   *intern;
3074:
3075:   if (zend_parse_parameters_none() == FAILURE) {
3076:     return;
3077:   }
3078:
3079:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3080:
3081:   if (Z_TYPE(intern->u.caching.zchildren) != IS_UNDEF) {
3082:     zval *value = &intern->u.caching.zchildren;
3083:
3084:     ZVAL_DEREF(value);
3085:     ZVAL_COPY(return_value, value);
3086:   } else {
3087:     RETURN_NULL();
3088:   }
3089: } /* }}} */
3090:
3091: ZEND_BEGIN_ARG_INFO_EX(arginfo_caching_rec_it___construct, 0, ZEND_RETURN_VALUE, 1)
3092:   ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
3093:   ZEND_ARG_INFO(0, flags)
3094: ZEND_END_ARG_INFO();
3095:
3096: static const zend_function_entry spl_funcs_RecursiveCachingIterator[] = {
3097:   SPL_ME(RecursiveCachingIterator, __construct,   arginfo_caching_rec_it___construct, ZEND_ACC_PUBLIC)
3098:   SPL_ME(RecursiveCachingIterator, hasChildren,   arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3099:   SPL_ME(RecursiveCachingIterator, getChildren,   arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3100:   PHP_FE_END
3101: };
3102:
3103: /* {{{ proto void IteratorIterator::__construct(Traversable it)
3104:    Create an iterator from anything that is traversable */
3105: SPL_METHOD(IteratorIterator, __construct)
3106: {
3107:   spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_IteratorIterator, zend_ce_traversable, DIT_IteratorIterator);
3108: } /* }}} */
3109:
3110: ZEND_BEGIN_ARG_INFO(arginfo_iterator_it___construct, 0)
3111:   ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
3112: ZEND_END_ARG_INFO();
3113:
3114: static const zend_function_entry spl_funcs_IteratorIterator[] = {
3115:   SPL_ME(IteratorIterator, __construct,      arginfo_iterator_it___construct, ZEND_ACC_PUBLIC)
3116:   SPL_ME(dual_it,          rewind,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3117:   SPL_ME(dual_it,          valid,            arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3118:   SPL_ME(dual_it,          key,              arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3119:   SPL_ME(dual_it,          current,          arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3120:   SPL_ME(dual_it,          next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3121:   SPL_ME(dual_it,          getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3122:   PHP_FE_END
3123: };
3124:
3125: /* {{{ proto void NoRewindIterator::__construct(Iterator it)
3126:    Create an iterator from another iterator */
3127: SPL_METHOD(NoRewindIterator, __construct)
3128: {
3129:   spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_NoRewindIterator, zend_ce_iterator, DIT_NoRewindIterator);
3130: } /* }}} */
3131:
3132: /* {{{ proto void NoRewindIterator::rewind()
3133:    Prevent a call to inner iterators rewind() */
3134: SPL_METHOD(NoRewindIterator, rewind)
3135: {
3136:   if (zend_parse_parameters_none() == FAILURE) {
3137:     return;
3138:   }
3139:   /* nothing to do */
3140: } /* }}} */
3141:
3142: /* {{{ proto bool NoRewindIterator::valid()
3143:    Return inner iterators valid() */
3144: SPL_METHOD(NoRewindIterator, valid)
3145: {
3146:   spl_dual_it_object   *intern;
3147:
3148:   if (zend_parse_parameters_none() == FAILURE) {
3149:     return;
3150:   }
3151:
3152:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3153:   RETURN_BOOL(intern->inner.iterator->funcs->valid(intern->inner.iterator) == SUCCESS);
3154: } /* }}} */
3155:
3156: /* {{{ proto mixed NoRewindIterator::key()
3157:    Return inner iterators key() */
3158: SPL_METHOD(NoRewindIterator, key)
3159: {
3160:   spl_dual_it_object   *intern;
3161:
3162:   if (zend_parse_parameters_none() == FAILURE) {
3163:     return;
3164:   }
3165:
3166:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3167:
3168:   if (intern->inner.iterator->funcs->get_current_key) {
3169:     intern->inner.iterator->funcs->get_current_key(intern->inner.iterator, return_value);
3170:   } else {
3171:     RETURN_NULL();
3172:   }
3173: } /* }}} */
3174:
3175: /* {{{ proto mixed NoRewindIterator::current()
3176:    Return inner iterators current() */
3177: SPL_METHOD(NoRewindIterator, current)
3178: {
3179:   spl_dual_it_object   *intern;
3180:   zval *data;
3181:
3182:   if (zend_parse_parameters_none() == FAILURE) {
3183:     return;
3184:   }
3185:
3186:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3187:   data = intern->inner.iterator->funcs->get_current_data(intern->inner.iterator);
3188:   if (data) {
3189:     ZVAL_DEREF(data);
3190:     ZVAL_COPY(return_value, data);
3191:   }
3192: } /* }}} */
3193:
3194: /* {{{ proto void NoRewindIterator::next()
3195:    Return inner iterators next() */
3196: SPL_METHOD(NoRewindIterator, next)
3197: {
3198:   spl_dual_it_object   *intern;
3199:
3200:   if (zend_parse_parameters_none() == FAILURE) {
3201:     return;
3202:   }
3203:
3204:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3205:   intern->inner.iterator->funcs->move_forward(intern->inner.iterator);
3206: } /* }}} */
3207:
3208: ZEND_BEGIN_ARG_INFO(arginfo_norewind_it___construct, 0)
3209:   ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
3210: ZEND_END_ARG_INFO();
3211:
3212: static const zend_function_entry spl_funcs_NoRewindIterator[] = {
3213:   SPL_ME(NoRewindIterator, __construct,      arginfo_norewind_it___construct, ZEND_ACC_PUBLIC)
3214:   SPL_ME(NoRewindIterator, rewind,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3215:   SPL_ME(NoRewindIterator, valid,            arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3216:   SPL_ME(NoRewindIterator, key,              arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3217:   SPL_ME(NoRewindIterator, current,          arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3218:   SPL_ME(NoRewindIterator, next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3219:   SPL_ME(dual_it,          getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3220:   PHP_FE_END
3221: };
3222:
3223: /* {{{ proto void InfiniteIterator::__construct(Iterator it)
3224:    Create an iterator from another iterator */
3225: SPL_METHOD(InfiniteIterator, __construct)
3226: {
3227:   spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_InfiniteIterator, zend_ce_iterator, DIT_InfiniteIterator);
3228: } /* }}} */
3229:
3230: /* {{{ proto void InfiniteIterator::next()
3231:    Prevent a call to inner iterators rewind() (internally the current data will be fetched if valid()) */
3232: SPL_METHOD(InfiniteIterator, next)
3233: {
3234:   spl_dual_it_object   *intern;
3235:
3236:   if (zend_parse_parameters_none() == FAILURE) {
3237:     return;
3238:   }
3239:
3240:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3241:
3242:   spl_dual_it_next(intern, 1);
3243:   if (spl_dual_it_valid(intern) == SUCCESS) {
3244:     spl_dual_it_fetch(intern, 0);
3245:   } else {
3246:     spl_dual_it_rewind(intern);
3247:     if (spl_dual_it_valid(intern) == SUCCESS) {
3248:       spl_dual_it_fetch(intern, 0);
3249:     }
3250:   }
3251: } /* }}} */
3252:
3253: static const zend_function_entry spl_funcs_InfiniteIterator[] = {
3254:   SPL_ME(InfiniteIterator, __construct,      arginfo_norewind_it___construct, ZEND_ACC_PUBLIC)
3255:   SPL_ME(InfiniteIterator, next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3256:   PHP_FE_END
3257: };
3258:
3259: /* {{{ proto void EmptyIterator::rewind()
3260:    Does nothing  */
3261: SPL_METHOD(EmptyIterator, rewind)
3262: {
3263:   if (zend_parse_parameters_none() == FAILURE) {
3264:     return;
3265:   }
3266: } /* }}} */
3267:
3268: /* {{{ proto false EmptyIterator::valid()
3269:    Return false */
3270: SPL_METHOD(EmptyIterator, valid)
3271: {
3272:   if (zend_parse_parameters_none() == FAILURE) {
3273:     return;
3274:   }
3275:   RETURN_FALSE;
3276: } /* }}} */
3277:
3278: /* {{{ proto void EmptyIterator::key()
3279:    Throws exception BadMethodCallException */
3280: SPL_METHOD(EmptyIterator, key)
3281: {
3282:   if (zend_parse_parameters_none() == FAILURE) {
3283:     return;
3284:   }
3285:   zend_throw_exception(spl_ce_BadMethodCallException, "Accessing the key of an EmptyIterator", 0);
3286: } /* }}} */
3287:
3288: /* {{{ proto void EmptyIterator::current()
3289:    Throws exception BadMethodCallException */
3290: SPL_METHOD(EmptyIterator, current)
3291: {
3292:   if (zend_parse_parameters_none() == FAILURE) {
3293:     return;
3294:   }
3295:   zend_throw_exception(spl_ce_BadMethodCallException, "Accessing the value of an EmptyIterator", 0);
3296: } /* }}} */
3297:
3298: /* {{{ proto void EmptyIterator::next()
3299:    Does nothing */
3300: SPL_METHOD(EmptyIterator, next)
3301: {
3302:   if (zend_parse_parameters_none() == FAILURE) {
3303:     return;
3304:   }
3305: } /* }}} */
3306:
3307: static const zend_function_entry spl_funcs_EmptyIterator[] = {
3308:   SPL_ME(EmptyIterator, rewind,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3309:   SPL_ME(EmptyIterator, valid,            arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3310:   SPL_ME(EmptyIterator, key,              arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3311:   SPL_ME(EmptyIterator, current,          arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3312:   SPL_ME(EmptyIterator, next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3313:   PHP_FE_END
3314: };
3315:
3316: int spl_append_it_next_iterator(spl_dual_it_object *intern) /* {{{*/
3317: {
3318:   spl_dual_it_free(intern);
3319:
3320:   if (!Z_ISUNDEF(intern->inner.zobject)) {
3321:     zval_ptr_dtor(&intern->inner.zobject);
3322:     ZVAL_UNDEF(&intern->inner.zobject);
3323:     intern->inner.ce = NULL;
3324:     if (intern->inner.iterator) {
3325:       zend_iterator_dtor(intern->inner.iterator);
3326:       intern->inner.iterator = NULL;
3327:     }
3328:   }
3329:   if (intern->u.append.iterator->funcs->valid(intern->u.append.iterator) == SUCCESS) {
3330:     zval *it;
3331:
3332:     it  = intern->u.append.iterator->funcs->get_current_data(intern->u.append.iterator);
3333:     ZVAL_COPY(&intern->inner.zobject, it);
3334:     intern->inner.ce = Z_OBJCE_P(it);
3335:     intern->inner.iterator = intern->inner.ce->get_iterator(intern->inner.ce, it, 0);
3336:     spl_dual_it_rewind(intern);
3337:     return SUCCESS;
3338:   } else {
3339:     return FAILURE;
3340:   }
3341: } /* }}} */
3342:
3343: void spl_append_it_fetch(spl_dual_it_object *intern) /* {{{*/
3344: {
3345:   while (spl_dual_it_valid(intern) != SUCCESS) {
3346:     intern->u.append.iterator->funcs->move_forward(intern->u.append.iterator);
3347:     if (spl_append_it_next_iterator(intern) != SUCCESS) {
3348:       return;
3349:     }
3350:   }
3351:   spl_dual_it_fetch(intern, 0);
3352: } /* }}} */
3353:
3354: void spl_append_it_next(spl_dual_it_object *intern) /* {{{ */
3355: {
3356:   if (spl_dual_it_valid(intern) == SUCCESS) {
3357:     spl_dual_it_next(intern, 1);
3358:   }
3359:   spl_append_it_fetch(intern);
3360: } /* }}} */
3361:
3362: /* {{{ proto void AppendIterator::__construct()
```

```
3363:    Create an AppendIterator */
3364: SPL_METHOD(AppendIterator, __construct)
3365: {
3366:   spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_AppendIterator, zend_ce_iterator, DIT_AppendIterator);
3367: } /* }}} */
3368:
3369: /* {{{ proto void AppendIterator::append(Iterator it)
3370:    Append an iterator */
3371: SPL_METHOD(AppendIterator, append)
3372: {
3373:   spl_dual_it_object   *intern;
3374:   zval *it;
3375:
3376:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3377:
3378:   if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "O", &it, zend_ce_iterator) == FAILURE) {
3379:     return;
3380:   }
3381:   if (intern->u.append.iterator->funcs->valid(intern->u.append.iterator) == SUCCESS && spl_dual_it_valid(intern) != SUCCESS) {
3382:     spl_array_iterator_append(&intern->u.append.zarrayit, it);
3383:     intern->u.append.iterator->funcs->move_forward(intern->u.append.iterator);
3384:   }else{
3385:     spl_array_iterator_append(&intern->u.append.zarrayit, it);
3386:   }
3387:
3388:   if (!intern->inner.iterator || spl_dual_it_valid(intern) != SUCCESS) {
3389:     if (intern->u.append.iterator->funcs->valid(intern->u.append.iterator) != SUCCESS) {
3390:       intern->u.append.iterator->funcs->rewind(intern->u.append.iterator);
3391:     }
3392:     do {
3393:       spl_append_it_next_iterator(intern);
3394:     } while (Z_OBJ(intern->inner.zobject) != Z_OBJ_P(it));
3395:     spl_append_it_fetch(intern);
3396:   }
3397: } /* }}} */
3398:
3399: /* {{{ proto mixed AppendIterator::current()
3400:    Get the current element value */
3401: SPL_METHOD(AppendIterator, current)
3402: {
3403:   spl_dual_it_object   *intern;
3404:
3405:   if (zend_parse_parameters_none() == FAILURE) {
3406:     return;
3407:   }
3408:
3409:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3410:
3411:   spl_dual_it_fetch(intern, 1);
3412:   if (Z_TYPE(intern->current.data) != IS_UNDEF) {
3413:     zval *value = &intern->current.data;
3414:
3415:     ZVAL_DEREF(value);
3416:     ZVAL_COPY(return_value, value);
3417:   } else {
3418:     RETURN_NULL();
3419:   }
3420: } /* }}} */
3421:
3422: /* {{{ proto void AppendIterator::rewind()
3423:    Rewind to the first iterator and rewind the first iterator, too */
3424: SPL_METHOD(AppendIterator, rewind)
3425: {
3426:   spl_dual_it_object   *intern;
3427:
3428:   if (zend_parse_parameters_none() == FAILURE) {
3429:     return;
3430:   }
3431:
3432:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3433:
3434:   intern->u.append.iterator->funcs->rewind(intern->u.append.iterator);
3435:   if (spl_append_it_next_iterator(intern) == SUCCESS) {
3436:     spl_append_it_fetch(intern);
3437:   }
3438: } /* }}} */
3439:
3440: /* {{{ proto bool AppendIterator::valid()
3441:    Check if the current state is valid */
3442: SPL_METHOD(AppendIterator, valid)
3443: {
3444:   spl_dual_it_object   *intern;
3445:
3446:   if (zend_parse_parameters_none() == FAILURE) {
3447:     return;
3448:   }
3449:
3450:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3451:
3452:   RETURN_BOOL(Z_TYPE(intern->current.data) != IS_UNDEF);
3453: } /* }}} */
3454:
3455: /* {{{ proto void AppendIterator::next()
3456:    Forward to next element */
3457: SPL_METHOD(AppendIterator, next)
3458: {
3459:   spl_dual_it_object   *intern;
3460:
3461:   if (zend_parse_parameters_none() == FAILURE) {
3462:     return;
3463:   }
3464:
3465:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3466:
3467:   spl_append_it_next(intern);
3468: } /* }}} */
3469:
3470: /* {{{ proto int AppendIterator::getIteratorIndex()
3471:    Get index of iterator */
3472: SPL_METHOD(AppendIterator, getIteratorIndex)
3473: {
3474:   spl_dual_it_object   *intern;
3475:
3476:   if (zend_parse_parameters_none() == FAILURE) {
3477:     return;
3478:   }
3479:
3480:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3481:
3482:   APPENDIT_CHECK_CTOR(intern);
3483:   spl_array_iterator_key(&intern->u.append.zarrayit, return_value);
3484: } /* }}} */
3485:
3486: /* {{{ proto ArrayIterator AppendIterator::getArrayIterator()
3487:    Get access to inner ArrayIterator */
3488: SPL_METHOD(AppendIterator, getArrayIterator)
3489: {
3490:   spl_dual_it_object   *intern;
3491:   zval *value;
3492:
3493:   if (zend_parse_parameters_none() == FAILURE) {
3494:     return;
3495:   }
3496:
3497:   SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3498:
3499:   value = &intern->u.append.zarrayit;
3500:   ZVAL_DEREF(value);
3501:   ZVAL_COPY(return_value, value);
3502: } /* }}} */
3503:
3504: ZEND_BEGIN_ARG_INFO(arginfo_append_it_append, 0)
3505:   ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
3506: ZEND_END_ARG_INFO();
3507:
3508: static const zend_function_entry spl_funcs_AppendIterator[] = {
3509:   SPL_ME(AppendIterator, __construct,      arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3510:   SPL_ME(AppendIterator, append,           arginfo_append_it_append, ZEND_ACC_PUBLIC)
3511:   SPL_ME(AppendIterator, rewind,           arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3512:   SPL_ME(AppendIterator, valid,            arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3513:   SPL_ME(dual_it,        key,              arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3514:   SPL_ME(AppendIterator, current,          arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3515:   SPL_ME(AppendIterator, next,             arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3516:   SPL_ME(dual_it,        getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3517:   SPL_ME(AppendIterator, getIteratorIndex, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3518:   SPL_ME(AppendIterator, getArrayIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
3519:   PHP_FE_END
3520: };
3521:
3522: PHPAPI int spl_iterator_apply(zval *obj, spl_iterator_apply_func_t apply_func, void *puser)
3523: {
3524:   zend_object_iterator  *iter;
3525:   zend_class_entry      *ce = Z_OBJCE_P(obj);
3526:
3527:   iter = ce->get_iterator(ce, obj, 0);
3528:
3529:   if (EG(exception)) {
3530:     goto done;
3531:   }
3532:
3533:   iter->index = 0;
3534:   if (iter->funcs->rewind) {
3535:     iter->funcs->rewind(iter);
3536:     if (EG(exception)) {
3537:       goto done;
3538:     }
3539:   }
3540:
3541:   while (iter->funcs->valid(iter) == SUCCESS) {
3542:     if (EG(exception)) {
3543:       goto done;
3544:     }
3545:     if (apply_func(iter, puser) == ZEND_HASH_APPLY_STOP || EG(exception)) {
3546:       goto done;
3547:     }
3548:     iter->index++;
3549:     iter->funcs->move_forward(iter);
3550:     if (EG(exception)) {
3551:       goto done;
3552:     }
3553:   }
3554:
3555: done:
3556:   if (iter) {
3557:     zend_iterator_dtor(iter);
3558:   }
3559:   return EG(exception) ? FAILURE : SUCCESS;
3560: }
3561: /* }}} */
3562:
3563: static int spl_iterator_to_array_apply(zend_object_iterator *iter, void *puser) /* {{{ */
3564: {
3565:   zval *data, *return_value = (zval*)puser;
3566:
3567:   data = iter->funcs->get_current_data(iter);
3568:   if (EG(exception)) {
3569:     return ZEND_HASH_APPLY_STOP;
3570:   }
3571:   if (data == NULL) {
3572:     return ZEND_HASH_APPLY_STOP;
3573:   }
3574:   if (iter->funcs->get_current_key) {
3575:     zval key;
3576:     iter->funcs->get_current_key(iter, &key);
3577:     if (EG(exception)) {
3578:       return ZEND_HASH_APPLY_STOP;
3579:     }
3580:     array_set_zval_key(Z_ARRVAL_P(return_value), &key, data);
3581:     zval_ptr_dtor(&key);
3582:   } else {
3583:     Z_TRY_ADDREF_P(data);
3584:     add_next_index_zval(return_value, data);
3585:   }
3586:   return ZEND_HASH_APPLY_KEEP;
3587: }
3588: /* }}} */
3589:
3590: static int spl_iterator_to_values_apply(zend_object_iterator *iter, void *puser) /* {{{ */
3591: {
3592:   zval *data, *return_value = (zval*)puser;
3593:
3594:   data = iter->funcs->get_current_data(iter);
3595:   if (EG(exception)) {
3596:     return ZEND_HASH_APPLY_STOP;
3597:   }
3598:   if (data == NULL) {
3599:     return ZEND_HASH_APPLY_STOP;
3600:   }
3601:   Z_TRY_ADDREF_P(data);
3602:   add_next_index_zval(return_value, data);
3603:   return ZEND_HASH_APPLY_KEEP;
3604: }
3605: /* }}} */
3606:
3607: /* {{{ proto array iterator_to_array(Traversable it [, bool use_keys = true])
3608:    Copy the iterator into an array */
3609: PHP_FUNCTION(iterator_to_array)
3610: {
3611:   zval  *obj;
3612:   zend_bool use_keys = 1;
3613:
3614:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "O|b", &obj, zend_ce_traversable, &use_keys) == FAILURE) {
3615:     RETURN_FALSE;
3616:   }
3617:
3618:   array_init(return_value);
3619:
3620:   if (spl_iterator_apply(obj, use_keys ? spl_iterator_to_array_apply : spl_iterator_to_values_apply, (void*)return_value) != SUCCESS) {
3621:     zval_ptr_dtor(return_value);
3622:     RETURN_NULL();
3623:   }
3624: } /* }}} */
3625:
3626: static int spl_iterator_count_apply(zend_object_iterator *iter, void *puser) /* {{{ */
3627: {
3628:   (*(zend_long*)puser)++;
3629:   return ZEND_HASH_APPLY_KEEP;
3630: }
3631: /* }}} */
3632:
3633: /* {{{ proto int iterator_count(Traversable it)
3634:    Count the elements in an iterator */
3635: PHP_FUNCTION(iterator_count)
3636: {
3637:   zval  *obj;
3638:   zend_long  count = 0;
3639:
3640:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, zend_ce_traversable) == FAILURE) {
3641:     RETURN_FALSE;
3642:   }
3643:
3644:   if (spl_iterator_apply(obj, spl_iterator_count_apply, (void*)&count) == SUCCESS) {
3645:     RETURN_LONG(count);
3646:   }
3647: }
3648: /* }}} */
3649:
3650: typedef struct {
3651:   zval                 *obj;
3652:   zval                 *args;
3653:   zend_long            count;
3654:   zend_fcall_info      fci;
3655:   zend_fcall_info_cache  fcc;
3656:   spl_iterator_apply_info;
3657:
3658: static int spl_iterator_func_apply(zend_object_iterator *iter, void *puser) /* {{{ */
3659: {
3660:   zval retval;
3661:   spl_iterator_apply_info  *apply_info = (spl_iterator_apply_info*)puser;
3662:   int result;
3663:
3664:   apply_info->count++;
3665:   zend_fcall_info_call(&apply_info->fci, &apply_info->fcc, &retval, NULL);
3666:   if (Z_TYPE(retval) != IS_UNDEF) {
3667:     result = zend_is_true(&retval) ? ZEND_HASH_APPLY_KEEP : ZEND_HASH_APPLY_STOP;
3668:     zval_ptr_dtor(&retval);
3669:   } else {
3670:     result = ZEND_HASH_APPLY_STOP;
3671:   }
3672:   return result;
3673: }
3674: /* }}} */
3675:
3676: /* {{{ proto int iterator_apply(Traversable it, mixed function [, mixed params])
3677:    Calls a function for every element in an iterator */
3678: PHP_FUNCTION(iterator_apply)
3679: {
3680:   spl_iterator_apply_info  apply_info;
3681:
3682:   apply_info.args = NULL;
3683:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "Of|a!", &apply_info.obj, zend_ce_traversable, &apply_info.fci, &apply_info.fcc, &apply_info.args) == FAIL
URE) {
3684:     return;
3685:   }
3686:
3687:   apply_info.count = 0;
3688:   zend_fcall_info_args(&apply_info.fci, apply_info.args);
3689:   if (spl_iterator_apply(apply_info.obj, spl_iterator_func_apply, (void*)&apply_info) == SUCCESS) {
3690:     RETVAL_LONG(apply_info.count);
3691:   } else {
3692:     RETVAL_FALSE;
3693:   }
3694:   zend_fcall_info_args(&apply_info.fci, NULL);
3695: }
3696: /* }}} */
3697:
3698: static const zend_function_entry spl_funcs_OuterIterator[] = {
3699:   SPL_ABSTRACT_ME(OuterIterator, getInnerIterator,   arginfo_recursive_it_void)
3700:   PHP_FE_END
3701: };
3702:
3703: /* {{{ PHP_MINIT_FUNCTION(spl_iterators)
3704:  */
3705: PHP_MINIT_FUNCTION(spl_iterators)
3706: {
3707:   REGISTER_SPL_INTERFACE(RecursiveIterator);
3708:   REGISTER_SPL_ITERATOR(RecursiveIterator);
3709:
3710:   REGISTER_SPL_STD_CLASS_EX(RecursiveIteratorIterator, spl_RecursiveIteratorIterator_new, spl_funcs_RecursiveIteratorIterator);
3711:   REGISTER_SPL_ITERATOR(RecursiveIteratorIterator);
3712:
3713:   memcpy(&spl_handlers_rec_it_it, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
3714:   spl_handlers_rec_it_it.offset = XtOffsetOf(spl_recursive_it_object, std);
3715:   spl_handlers_rec_it_it.get_method = spl_recursive_it_get_method;
3716:   spl_handlers_rec_it_it.clone_obj = NULL;
3717:   spl_handlers_rec_it_it.dtor_obj = spl_RecursiveIteratorIterator_dtor;
3718:   spl_handlers_rec_it_it.free_obj = spl_RecursiveIteratorIterator_free_storage;
3719:
3720:   memcpy(&spl_handlers_dual_it, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
3721:   spl_handlers_dual_it.offset = XtOffsetOf(spl_dual_it_object, std);
3722:   spl_handlers_dual_it.get_method = spl_dual_it_get_method;
3723:   /*spl_handlers_dual_it.call_method = spl_dual_it_call_method;*/
3724:   spl_handlers_dual_it.clone_obj = NULL;
3725:   spl_handlers_dual_it.dtor_obj = spl_dual_it_dtor;
3726:   spl_handlers_dual_it.free_obj = spl_dual_it_free_storage;
3727:
3728:   spl_ce_RecursiveIteratorIterator->get_iterator = spl_recursive_it_get_iterator;
3729:   spl_ce_RecursiveIteratorIterator->iterator_funcs.funcs = &spl_recursive_it_iterator_funcs;
3730:
3731:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveIteratorIterator, "LEAVES_ONLY",     RIT_LEAVES_ONLY);
3732:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveIteratorIterator, "SELF_FIRST",      RIT_SELF_FIRST);
3733:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveIteratorIterator, "CHILD_FIRST",     RIT_CHILD_FIRST);
3734:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveIteratorIterator, "CATCH_GET_CHILD", RIT_CATCH_GET_CHILD);
3735:
3736:   REGISTER_SPL_INTERFACE(OuterIterator);
3737:   REGISTER_SPL_ITERATOR(OuterIterator);
```

```
3738:
3739:   REGISTER_SPL_STD_CLASS_EX(IteratorIterator, spl_dual_it_new, spl_funcs_IteratorIterator);
3740:   REGISTER_SPL_ITERATOR(IteratorIterator);
3741:   REGISTER_SPL_IMPLEMENTS(IteratorIterator, OuterIterator);
3742:
3743:   REGISTER_SPL_SUB_CLASS_EX(FilterIterator, IteratorIterator, spl_dual_it_new, spl_funcs_FilterIterator);
3744:   spl_ce_FilterIterator->ce_flags |= ZEND_ACC_EXPLICIT_ABSTRACT_CLASS;
3745:
3746:   REGISTER_SPL_SUB_CLASS_EX(RecursiveFilterIterator, FilterIterator, spl_dual_it_new, spl_funcs_RecursiveFilterIterator);
3747:   REGISTER_SPL_IMPLEMENTS(RecursiveFilterIterator, RecursiveIterator);
3748:
3749:   REGISTER_SPL_SUB_CLASS_EX(CallbackFilterIterator, FilterIterator, spl_dual_it_new, spl_funcs_CallbackFilterIterator);
3750:
3751:   REGISTER_SPL_SUB_CLASS_EX(RecursiveCallbackFilterIterator, CallbackFilterIterator, spl_dual_it_new, spl_funcs_RecursiveCallbackFilterIterator);
3752:   REGISTER_SPL_IMPLEMENTS(RecursiveCallbackFilterIterator, RecursiveIterator);
3753:
3754:
3755:   REGISTER_SPL_SUB_CLASS_EX(ParentIterator, RecursiveFilterIterator, spl_dual_it_new, spl_funcs_ParentIterator);
3756:
3757:   REGISTER_SPL_INTERFACE(SeekableIterator);
3758:   REGISTER_SPL_ITERATOR(SeekableIterator);
3759:
3760:   REGISTER_SPL_SUB_CLASS_EX(LimitIterator, IteratorIterator, spl_dual_it_new, spl_funcs_LimitIterator);
3761:
3762:   REGISTER_SPL_SUB_CLASS_EX(CachingIterator, IteratorIterator, spl_dual_it_new, spl_funcs_CachingIterator);
3763:   REGISTER_SPL_IMPLEMENTS(CachingIterator, ArrayAccess);
3764:   REGISTER_SPL_IMPLEMENTS(CachingIterator, Countable);
3765:
3766:   REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "CALL_TOSTRING",       CIT_CALL_TOSTRING);
3767:   REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "CATCH_GET_CHILD",     CIT_CATCH_GET_CHILD);
3768:   REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_KEY",    CIT_TOSTRING_USE_KEY);
3769:   REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_CURRENT", CIT_TOSTRING_USE_CURRENT);
3770:   REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_INNER",  CIT_TOSTRING_USE_INNER);
3771:   REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "FULL_CACHE",          CIT_FULL_CACHE);
3772:
3773:   REGISTER_SPL_SUB_CLASS_EX(RecursiveCachingIterator, CachingIterator, spl_dual_it_new, spl_funcs_RecursiveCachingIterator);
3774:   REGISTER_SPL_IMPLEMENTS(RecursiveCachingIterator, RecursiveIterator);
3775:
3776:   REGISTER_SPL_SUB_CLASS_EX(NoRewindIterator, IteratorIterator, spl_dual_it_new, spl_funcs_NoRewindIterator);
3777:
3778:   REGISTER_SPL_SUB_CLASS_EX(AppendIterator, IteratorIterator, spl_dual_it_new, spl_funcs_AppendIterator);
3779:
3780:   REGISTER_SPL_IMPLEMENTS(RecursiveIteratorIterator, OuterIterator);
3781:
3782:   REGISTER_SPL_SUB_CLASS_EX(InfiniteIterator, IteratorIterator, spl_dual_it_new, spl_funcs_InfiniteIterator);
3783: #if HAVE_PCRE || HAVE_BUNDLED_PCRE
3784:   REGISTER_SPL_SUB_CLASS_EX(RegexIterator, FilterIterator, spl_dual_it_new, spl_funcs_RegexIterator);
3785:   REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "USE_KEY",     REGIT_USE_KEY);
3786:   REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "INVERT_MATCH",REGIT_INVERTED);
3787:   REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "MATCH",       REGIT_MODE_MATCH);
3788:   REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "GET_MATCH",   REGIT_MODE_GET_MATCH);
3789:   REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "ALL_MATCHES", REGIT_MODE_ALL_MATCHES);
3790:   REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "SPLIT",       REGIT_MODE_SPLIT);
3791:   REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "REPLACE",     REGIT_MODE_REPLACE);
3792:   REGISTER_SPL_PROPERTY(RegexIterator, "replacement", 0);
3793:   REGISTER_SPL_SUB_CLASS_EX(RecursiveRegexIterator, RegexIterator, spl_dual_it_new, spl_funcs_RecursiveRegexIterator);
3794:   REGISTER_SPL_IMPLEMENTS(RecursiveRegexIterator, RecursiveIterator);
3795: #else
3796:   spl_ce_RegexIterator = NULL;
3797:   spl_ce_RecursiveRegexIterator = NULL;
3798: #endif
3799:
3800:   REGISTER_SPL_STD_CLASS_EX(EmptyIterator, NULL, spl_funcs_EmptyIterator);
3801:   REGISTER_SPL_ITERATOR(EmptyIterator);
3802:
3803:   REGISTER_SPL_SUB_CLASS_EX(RecursiveTreeIterator, RecursiveIteratorIterator, spl_RecursiveTreeIterator_new, spl_funcs_RecursiveTreeIterator);
3804:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "BYPASS_CURRENT",      RTIT_BYPASS_CURRENT);
3805:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "BYPASS_KEY",          RTIT_BYPASS_KEY);
3806:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_LEFT",         0);
3807:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_MID_HAS_NEXT", 1);
3808:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_MID_LAST",     2);
3809:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_END_HAS_NEXT", 3);
3810:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_END_LAST",     4);
3811:   REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_RIGHT",        5);
3812:
3813:   return SUCCESS;
3814: }
3815: /* }}} */
3816:
3817: /*
3818:  * Local variables:
3819:  * tab-width: 4
3820:  * c-basic-offset: 4
3821:  * End:
3822:  * vim600: fdm=marker
3823:  * vim: noet sw=4 ts=4
3824:  */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Etienne Kneuss <colder@php.net>                             |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_DLLIST_H
#define SPL_DLLIST_H

#include "php.h"
#include "php_spl.h"

extern PHPAPI zend_class_entry *spl_ce_SplDoublyLinkedList;
extern PHPAPI zend_class_entry *spl_ce_SplQueue;
extern PHPAPI zend_class_entry *spl_ce_SplStack;

PHP_MINIT_FUNCTION(spl_dllist);

#endif /* SPL_DLLIST_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef PHP_FUNCTIONS_H
#define PHP_FUNCTIONS_H

#include "php.h"

typedef zend_object* (*create_object_func_t)(zend_class_entry *class_type);

#define REGISTER_SPL_STD_CLASS(class_name, obj_ctor) \
  spl_register_std_class(&spl_ce_ ## class_name, # class_name, obj_ctor, NULL);
#define REGISTER_SPL_STD_CLASS_EX(class_name, obj_ctor, funcs) \
  spl_register_std_class(&spl_ce_ ## class_name, # class_name, obj_ctor, funcs);
#define REGISTER_SPL_SUB_CLASS_EX(class_name, parent_class_name, obj_ctor, funcs) \
  spl_register_sub_class(&spl_ce_ ## class_name, spl_ce_ ## parent_class_name, # class_name, obj_ctor, funcs);
#define REGISTER_SPL_INTERFACE(class_name) \
  spl_register_interface(&spl_ce_ ## class_name, # class_name, spl_funcs_ ## class_name);
#define REGISTER_SPL_IMPLEMENTS(class_name, interface_name) \
  zend_class_implements(spl_ce_ ## class_name, 1, spl_ce_ ## interface_name);
#define REGISTER_SPL_ITERATOR(class_name) \
  zend_class_implements(spl_ce_ ## class_name, 1, zend_ce_iterator);
#define REGISTER_SPL_PROPERTY(class_name, prop_name, prop_flags) \
  spl_register_property(spl_ce_ ## class_name, prop_name, sizeof(prop_name)-1, prop_flags);
#define REGISTER_SPL_CLASS_CONST_LONG(class_name, const_name, value) \
  zend_declare_class_constant_long(spl_ce_ ## class_name, const_name, sizeof(const_name)-1, (zend_long)value);
void spl_register_std_class(zend_class_entry ** ppce, char * class_name, create_object_func_t ctor, const zend_function_entry * function_list);
void spl_register_sub_class(zend_class_entry ** ppce, zend_class_entry * parent_ce, char * class_name, create_object_func_t ctor, const zend_function_entry * function_list);
void spl_register_interface(zend_class_entry ** ppce, char * class_name, const zend_function_entry *functions);
void spl_register_property( zend_class_entry * class_entry, char *prop_name, int prop_name_len, int prop_flags);
/* sub: whether to allow subclasses/interfaces
   allow = 0: allow all classes and interfaces
   allow > 0: allow all that match and mask ce_flags
   allow < 0: disallow all that match and mask ce_flags
 */
void spl_add_class_name(zval * list, zend_class_entry * pce, int allow, int ce_flags);
void spl_add_interfaces(zval * list, zend_class_entry * pce, int allow, int ce_flags);
void spl_add_traits(zval * list, zend_class_entry * pce, int allow, int ce_flags);
int spl_add_classes(zend_class_entry *pce, zval *list, int sub, int allow, int ce_flags);
/* caller must efree(return) */
zend_string *spl_gen_private_prop_name(zend_class_entry *ce, char *prop_name, int prop_len);
#define SPL_ME(class_name, function_name, arg_info, flags) \
  PHP_ME( spl_ ## class_name, function_name, arg_info, flags)
#define SPL_ABSTRACT_ME(class_name, function_name, arg_info) \
  ZEND_ABSTRACT_ME( spl_ ## class_name, function_name, arg_info)
#define SPL_METHOD(class_name, function_name) \
  PHP_METHOD(spl_ ## class_name, function_name)
#define SPL_MA(class_name, function_name, alias_class, alias_function, arg_info, flags) \
  PHP_MALIAS(spl_ ## alias_class, function_name, alias_function, arg_info, flags)
#endif /* PHP_FUNCTIONS_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "php_main.h"
#include "ext/standard/info.h"
#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_array.h"
#include "spl_directory.h"
#include "spl_iterators.h"
#include "spl_exceptions.h"
#include "spl_observer.h"
#include "spl_dllist.h"
#include "spl_fixedarray.h"
#include "spl_heap.h"
#include "zend_exceptions.h"
#include "zend_interfaces.h"
#include "ext/standard/php_mt_rand.h"
#include "main/snprintf.h"

#ifdef COMPILE_DL_SPL
ZEND_GET_MODULE(spl)
#endif

ZEND_DECLARE_MODULE_GLOBALS(spl)

#define SPL_DEFAULT_FILE_EXTENSIONS ".inc,.php"

/* {{{ PHP_GINIT_FUNCTION
 */
static PHP_GINIT_FUNCTION(spl)
{
	spl_globals->autoload_extensions     = NULL;
	spl_globals->autoload_functions      = NULL;
	spl_globals->autoload_running        = 0;
}
/* }}} */

static zend_class_entry * spl_find_ce_by_name(zend_string *name, zend_bool autoload)
{
	zend_class_entry *ce;

	if (!autoload) {
		zend_string *lc_name = zend_string_tolower(name);

		ce = zend_hash_find_ptr(EG(class_table), lc_name);
		zend_string_free(lc_name);
	} else {
		ce = zend_lookup_class(name);
	}
	if (ce == NULL) {
		php_error_docref(NULL, E_WARNING, "Class %s does not exist%s", ZSTR_VAL(name), autoload ? " and could not be loaded" : "");
		return NULL;
	}

	return ce;
}

/* {{{ proto array class_parents(object instance [, bool autoload = true])
   Return an array containing the names of all parent classes */
PHP_FUNCTION(class_parents)
{
	zval *obj;
	zend_class_entry *parent_class, *ce;
	zend_bool autoload = 1;

	if (zend_parse_parameters(ZEND_NUM_ARGS(), "z|b", &obj, &autoload) == FAILURE) {
		RETURN_FALSE;
	}

	if (Z_TYPE_P(obj) != IS_OBJECT && Z_TYPE_P(obj) != IS_STRING) {
		php_error_docref(NULL, E_WARNING, "object or string expected");
		RETURN_FALSE;
	}

	if (Z_TYPE_P(obj) == IS_STRING) {
		if (NULL == (ce = spl_find_ce_by_name(Z_STR_P(obj), autoload))) {
			RETURN_FALSE;
		}
	} else {
		ce = Z_OBJCE_P(obj);
	}

	array_init(return_value);
	parent_class = ce->parent;
	while (parent_class) {
		spl_add_class_name(return_value, parent_class, 0, 0);
		parent_class = parent_class->parent;
	}
}
/* }}} */

/* {{{ proto array class_implements(mixed what [, bool autoload ])
   Return all classes and interfaces implemented by SPL */
PHP_FUNCTION(class_implements)
{
	zval *obj;
	zend_bool autoload = 1;
	zend_class_entry *ce;

	if (zend_parse_parameters(ZEND_NUM_ARGS(), "z|b", &obj, &autoload) == FAILURE) {
		RETURN_FALSE;
	}
	if (Z_TYPE_P(obj) != IS_OBJECT && Z_TYPE_P(obj) != IS_STRING) {
		php_error_docref(NULL, E_WARNING, "object or string expected");
		RETURN_FALSE;
	}

	if (Z_TYPE_P(obj) == IS_STRING) {
		if (NULL == (ce = spl_find_ce_by_name(Z_STR_P(obj), autoload))) {
			RETURN_FALSE;
		}
	} else {
		ce = Z_OBJCE_P(obj);
	}

	array_init(return_value);
	spl_add_interfaces(return_value, ce, 1, ZEND_ACC_INTERFACE);
}
/* }}} */

/* {{{ proto array class_uses(mixed what [, bool autoload ])
   Return all traits used by a class. */
PHP_FUNCTION(class_uses)
{
	zval *obj;
	zend_bool autoload = 1;
	zend_class_entry *ce;

	if (zend_parse_parameters(ZEND_NUM_ARGS(), "z|b", &obj, &autoload) == FAILURE) {
		RETURN_FALSE;
	}
	if (Z_TYPE_P(obj) != IS_OBJECT && Z_TYPE_P(obj) != IS_STRING) {
		php_error_docref(NULL, E_WARNING, "object or string expected");
		RETURN_FALSE;
	}

	if (Z_TYPE_P(obj) == IS_STRING) {
		if (NULL == (ce = spl_find_ce_by_name(Z_STR_P(obj), autoload))) {
			RETURN_FALSE;
		}
	} else {
		ce = Z_OBJCE_P(obj);
	}

	array_init(return_value);
	spl_add_traits(return_value, ce, 1, ZEND_ACC_TRAIT);
}
/* }}} */

#define SPL_ADD_CLASS(class_name, z_list, sub, allow, ce_flags) \
	spl_add_classes(spl_ce_ ## class_name, z_list, sub, allow, ce_flags)

#define SPL_LIST_CLASSES(z_list, sub, allow, ce_flags) \
	SPL_ADD_CLASS(AppendIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(ArrayIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(ArrayObject, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(BadFunctionCallException, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(BadMethodCallException, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(CachingIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(CallbackFilterIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(DirectoryIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(DomainException, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(EmptyIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(FilesystemIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(FilterIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(GlobIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(InfiniteIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(InvalidArgumentException, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(IteratorIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(LengthException, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(LimitIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(LogicException, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(MultipleIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(NoRewindIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(OuterIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(OutOfBoundsException, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(OutOfRangeException, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(OverflowException, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(ParentIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(RangeException, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(RecursiveArrayIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(RecursiveCachingIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(RecursiveCallbackFilterIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(RecursiveDirectoryIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(RecursiveFilterIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(RecursiveIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(RecursiveIteratorIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(RecursiveRegexIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(RecursiveTreeIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(RegexIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(RuntimeException, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SeekableIterator, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplDoublyLinkedList, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplFileInfo, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplFileObject, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplFixedArray, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplHeap, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplMinHeap, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplMaxHeap, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplObjectStorage, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplObserver, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplPriorityQueue, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplQueue, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplStack, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplSubject, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(SplTempFileObject, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(UnderflowException, z_list, sub, allow, ce_flags); \
	SPL_ADD_CLASS(UnexpectedValueException, z_list, sub, allow, ce_flags); \

/* {{{ proto array spl_classes()
   Return an array containing the names of all clsses and interfaces defined in SPL */
PHP_FUNCTION(spl_classes)
{
	array_init(return_value);

	SPL_LIST_CLASSES(return_value, 0, 0, 0)
}
/* }}} */

static int spl_autoload(zend_string *class_name, zend_string *lc_name, const char *ext, int ext_len) /* {{{ */
{
	char *class_file;
	int class_file_len;
	zval dummy;
	zend_file_handle file_handle;
	zend_op_array *new_op_array;
	zval result;
	int ret;

	class_file_len = (int)spprintf(&class_file, 0, "%s%.*s", ZSTR_VAL(lc_name), ext_len, ext);

#if DEFAULT_SLASH != '\\'
	{
		char *ptr = class_file;
		char *end = ptr + class_file_len;

		while ((ptr = memchr(ptr, '\\', (end - ptr))) != NULL) {
			*ptr = DEFAULT_SLASH;
		}
	}
#endif

	ret = php_stream_open_for_zend_ex(class_file, &file_handle, USE_PATH|STREAM_OPEN_FOR_INCLUDE);

	if (ret == SUCCESS) {
		zend_string *opened_path;
		if (!file_handle.opened_path) {
			file_handle.opened_path = zend_string_init(class_file, class_file_len, 0);
		}
		opened_path = zend_string_copy(file_handle.opened_path);
		ZVAL_NULL(&dummy);
		if (zend_hash_add(&EG(included_files), opened_path, &dummy)) {
			new_op_array = zend_compile_file(&file_handle, ZEND_REQUIRE);
			zend_destroy_file_handle(&file_handle);
		} else {
			new_op_array = NULL;
			zend_file_handle_dtor(&file_handle);
		}
		zend_string_release(opened_path);
		if (new_op_array) {
			ZVAL_UNDEF(&result);
			zend_execute(new_op_array, &result);

			destroy_op_array(new_op_array);
			efree(new_op_array);
			if (!EG(exception)) {
				zval_ptr_dtor(&result);
			}

			efree(class_file);
			return zend_hash_exists(EG(class_table), lc_name);
		}
	}
	efree(class_file);
	return 0;
} /* }}} */

/* {{{ proto void spl_autoload(string class_name [, string file_extensions])
   Default implementation for __autoload() */
PHP_FUNCTION(spl_autoload)
{
	int pos_len, posl_len;
	char *pos, *posl;
	zend_string *class_name, *lc_name, *file_exts = SPL_G(autoload_extensions);

	if (zend_parse_parameters(ZEND_NUM_ARGS(), "S|S", &class_name, &file_exts) == FAILURE) {
		RETURN_FALSE;
	}

	if (file_exts == NULL) { /* autoload_extensions is not initialized, set to defaults */
		pos = SPL_DEFAULT_FILE_EXTENSIONS;
		pos_len = sizeof(SPL_DEFAULT_FILE_EXTENSIONS) - 1;
	} else {
		pos = ZSTR_VAL(file_exts);
		pos_len = (int)ZSTR_LEN(file_exts);
	}

	lc_name = zend_string_tolower(class_name);
	while (pos && *pos && !EG(exception)) {
		posl = strchr(pos, ',');
		if (posl) {
			posl_len = (int)(posl - pos);
		} else {
			posl_len = pos_len;
		}
		if (spl_autoload(class_name, lc_name, pos, posl_len)) {
			break; /* loaded */
		}
		pos = posl ? posl + 1 : NULL;
		pos_len = posl? pos_len - posl_len - 1 : 0;
	}
	zend_string_free(lc_name);
} /* }}} */

/* {{{ proto string spl_autoload_extensions([string file_extensions])
   Register and return default file extensions for spl_autoload */
PHP_FUNCTION(spl_autoload_extensions)
{
	zend_string *file_exts = NULL;

	if (zend_parse_parameters(ZEND_NUM_ARGS(), "|S", &file_exts) == FAILURE) {
		return;
	}
	if (file_exts) {
		if (SPL_G(autoload_extensions)) {
			zend_string_release(SPL_G(autoload_extensions));
		}
		SPL_G(autoload_extensions) = zend_string_copy(file_exts);
	}

	if (SPL_G(autoload_extensions) == NULL) {
		RETURN_STRINGL(SPL_DEFAULT_FILE_EXTENSIONS, sizeof(SPL_DEFAULT_FILE_EXTENSIONS) - 1);
	} else {
		zend_string_addref(SPL_G(autoload_extensions));
		RETURN_STR(SPL_G(autoload_extensions));
	}
} /* }}} */

typedef struct {
	zend_function *func_ptr;
	zval obj;
	zval closure;
	zend_class_entry *ce;
} autoload_func_info;

static void autoload_func_info_dtor(zval *element)
{
	autoload_func_info *alfi = (autoload_func_info*)Z_PTR_P(element);
	if (!Z_ISUNDEF(alfi->obj)) {
		zval_ptr_dtor(&alfi->obj);
	}
```

```
377:   if (alfi->func_ptr &&
378:     UNEXPECTED(alfi->func_ptr->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
379:     zend_string_release(alfi->func_ptr->common.function_name);
380:     zend_free_trampoline(alfi->func_ptr);
381:   }
382:   if (!Z_ISUNDEF(alfi->closure)) {
383:     zval_ptr_dtor(&alfi->closure);
384:   }
385:   efree(alfi);
386: }
387:
388: /* {{{ proto void spl_autoload_call(string class_name)
389:    Try all registerd autoload function to load the requested class */
390: PHP_FUNCTION(spl_autoload_call)
391: {
392:   zval *class_name, retval;
393:   zend_string *lc_name, *func_name;
394:   autoload_func_info *alfi;
395:
396:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &class_name) == FAILURE || Z_TYPE_P(class_name) != IS_STRING) {
397:     return;
398:   }
399:
400:   if (SPL_G(autoload_functions)) {
401:     HashPosition pos;
402:     zend_ulong num_idx;
403:     zend_function *func;
404:     zend_fcall_info fci;
405:     zend_fcall_info_cache fcic;
406:     zend_class_entry *called_scope = zend_get_called_scope(execute_data);
407:     int l_autoload_running = SPL_G(autoload_running);
408:
409:     SPL_G(autoload_running) = 1;
410:     lc_name = zend_string_tolower(Z_STR_P(class_name));
411:
412:     fci.size = sizeof(fci);
413:     fci.retval = &retval;
414:     fci.param_count = 1;
415:     fci.params = class_name;
416:     fci.no_separation = 1;
417:
418:     ZVAL_UNDEF(&fci.function_name); /* Unused */
419:
420:     zend_hash_internal_pointer_reset_ex(SPL_G(autoload_functions), &pos);
421:     while (zend_hash_get_current_key_ex(SPL_G(autoload_functions), &func_name, &num_idx, &pos) == HASH_KEY_IS_STRING) {
422:       alfi = zend_hash_get_current_data_ptr_ex(SPL_G(autoload_functions), &pos);
423:       func = alfi->func_ptr;
424:       if (UNEXPECTED(func->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
425:         func = emalloc(sizeof(zend_op_array));
426:         memcpy(func, alfi->func_ptr, sizeof(zend_op_array));
427:         zend_string_addref(func->op_array.function_name);
428:       }
429:       ZVAL_UNDEF(&retval);
430:       fcic.function_handler = func;
431:       if (Z_ISUNDEF(alfi->obj)) {
432:         fci.object = NULL;
433:         fcic.object = NULL;
434:         fcic.calling_scope = alfi->ce;
435:         if (alfi->ce &&
436:             (!called_scope ||
437:              !instanceof_function(called_scope, alfi->ce))) {
438:           fcic.called_scope = alfi->ce;
439:         } else {
440:           fcic.called_scope = called_scope;
441:         }
442:       } else {
443:         fci.object = Z_OBJ(alfi->obj);
444:         fcic.object = Z_OBJ(alfi->obj);
445:         fcic.called_scope = Z_OBJCE(alfi->obj);
446:       }
447:
448:       zend_call_function(&fci, &fcic);
449:       zval_ptr_dtor(&retval);
450:
451:       if (EG(exception)) {
452:         break;
453:       }
454:
455:       if (pos + 1 == SPL_G(autoload_functions)->nNumUsed ||
456:           zend_hash_exists(EG(class_table), lc_name)) {
457:         break;
458:       }
459:       zend_hash_move_forward_ex(SPL_G(autoload_functions), &pos);
460:     }
461:     zend_string_release(lc_name);
462:     SPL_G(autoload_running) = l_autoload_running;
463:   } else {
464:     /* do not use or overwrite &EG(autoload_func) here */
465:     zend_call_method_with_1_params(NULL, NULL, NULL, "spl_autoload", NULL, class_name);
466:   }
467: } /* }}} */
468:
469: #define HT_MOVE_TAIL_TO_HEAD(ht)                    \
470:   do {                                              \
471:     Bucket tmp = (ht)->arData[(ht)->nNumUsed-1];    \
472:     memmove((ht)->arData + 1, (ht)->arData,         \
473:       sizeof(Bucket) * ((ht)->nNumUsed - 1));       \
474:     (ht)->arData[0] = tmp;                          \
475:     zend_hash_rehash(ht);                           \
476:   } while (0)
477:
478: /* {{{ proto bool spl_autoload_register([mixed autoload_function [, bool throw [, bool prepend]]])
479:    Register given function as __autoload() implementation */
480: PHP_FUNCTION(spl_autoload_register)
481: {
482:   zend_string *func_name;
483:   char *error = NULL;
484:   zend_string *lc_name;
485:   zval *zcallable = NULL;
486:   zend_bool do_throw = 1;
487:   zend_bool prepend  = 0;
488:   zend_function *spl_func_ptr;
489:   autoload_func_info alfi;
490:   zend_object *obj_ptr;
491:   zend_fcall_info_cache fcc;
492:
493:   if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "|zbb", &zcallable, &do_throw, &prepend) == FAILURE) {
494:     return;
495:   }
496:
497:   if (ZEND_NUM_ARGS()) {
498:     if (!zend_is_callable_ex(zcallable, NULL, IS_CALLABLE_STRICT, &func_name, &fcc, &error)) {
499:       alfi.ce = fcc.calling_scope;
500:       alfi.func_ptr = fcc.function_handler;
501:       obj_ptr = fcc.object;
502:       if (Z_TYPE_P(zcallable) == IS_ARRAY) {
503:         if (!obj_ptr && alfi.func_ptr && !(alfi.func_ptr->common.fn_flags & ZEND_ACC_STATIC)) {
504:           if (do_throw) {
505:             zend_throw_exception_ex(spl_ce_LogicException, 0, "Passed array specifies a non static method but no object (%s)", error);
506:           }
507:           if (error) {
508:             efree(error);
509:           }
510:           zend_string_release(func_name);
511:           RETURN_FALSE;
512:         } else if (do_throw) {
513:           zend_throw_exception_ex(spl_ce_LogicException, 0, "Passed array does not specify %s %smethod (%s)", alfi.func_ptr ? "a callable" : "an existi
ng", !obj_ptr ? "static " : "", error);
514:         }
515:         if (error) {
516:           efree(error);
517:         }
518:         zend_string_release(func_name);
519:         RETURN_FALSE;
520:       } else if (Z_TYPE_P(zcallable) == IS_STRING) {
521:         if (do_throw) {
522:           zend_throw_exception_ex(spl_ce_LogicException, 0, "Function '%s' not %s (%s)", ZSTR_VAL(func_name), alfi.func_ptr ? "callable" : "found", err
or);
523:         }
524:         if (error) {
525:           efree(error);
526:         }
527:         zend_string_release(func_name);
528:         RETURN_FALSE;
529:       } else {
530:         if (do_throw) {
531:           zend_throw_exception_ex(spl_ce_LogicException, 0, "Illegal value passed (%s)", error);
532:         }
533:         if (error) {
534:           efree(error);
535:         }
536:         zend_string_release(func_name);
537:         RETURN_FALSE;
538:       }
539:     } else if (fcc.function_handler->type == ZEND_INTERNAL_FUNCTION &&
540:           fcc.function_handler->internal_function.handler == zif_spl_autoload_call) {
541:       if (do_throw) {
542:         zend_throw_exception_ex(spl_ce_LogicException, 0, "Function spl_autoload_call() cannot be registered");
543:       }
544:       if (error) {
545:         efree(error);
546:       }
547:       zend_string_release(func_name);
548:       RETURN_FALSE;
549:     }
550:     alfi.ce = fcc.calling_scope;
551:     alfi.func_ptr = fcc.function_handler;
552:     obj_ptr = fcc.object;
553:     if (error) {
554:       efree(error);
555:     }
556:
557:     if (Z_TYPE_P(zcallable) == IS_OBJECT) {
558:       ZVAL_COPY(&alfi.closure, zcallable);
559:
560:       lc_name = zend_string_alloc(ZSTR_LEN(func_name) + sizeof(uint32_t), 0);
561:       zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name), ZSTR_LEN(func_name));
562:       memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(func_name), &Z_OBJ_HANDLE_P(zcallable), sizeof(uint32_t));
563:       ZSTR_VAL(lc_name)[ZSTR_LEN(lc_name)] = '\0';
564:     } else {
565:       ZVAL_UNDEF(&alfi.closure);
566:       /* Skip leading \ */
567:       if (ZSTR_VAL(func_name)[0] == '\\') {
568:         lc_name = zend_string_alloc(ZSTR_LEN(func_name) - 1, 0);
569:         zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name) + 1, ZSTR_LEN(func_name) - 1);
570:       } else {
571:         lc_name = zend_string_tolower(func_name);
572:       }
573:     }
574:     zend_string_release(func_name);
575:
576:     if (SPL_G(autoload_functions) && zend_hash_exists(SPL_G(autoload_functions), lc_name)) {
577:       if (!Z_ISUNDEF(alfi.closure)) {
578:         Z_DELREF_P(&alfi.closure);
579:       }
580:       goto skip;
581:     }
582:
583:     if (obj_ptr && !(alfi.func_ptr->common.fn_flags & ZEND_ACC_STATIC)) {
584:       /* add object id to the hash to ensure uniqueness, for more reference look at bug #40091 */
585:       lc_name = zend_string_extend(lc_name, ZSTR_LEN(lc_name) + sizeof(uint32_t), 0);
586:       memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(lc_name) - sizeof(uint32_t), &obj_ptr->handle, sizeof(uint32_t));
587:       ZSTR_VAL(lc_name)[ZSTR_LEN(lc_name)] = '\0';
588:       ZVAL_OBJ(&alfi.obj, obj_ptr);
589:       Z_ADDREF(alfi.obj);
590:     } else {
591:       ZVAL_UNDEF(&alfi.obj);
592:     }
593:
594:     if (!SPL_G(autoload_functions)) {
595:       ALLOC_HASHTABLE(SPL_G(autoload_functions));
596:       zend_hash_init(SPL_G(autoload_functions), 1, NULL, autoload_func_info_dtor, 0);
597:     }
598:
599:     spl_func_ptr = zend_hash_str_find_ptr(EG(function_table), "spl_autoload", sizeof("spl_autoload") - 1);
600:
601:     if (EG(autoload_func) == spl_func_ptr) { /* registered already, so we insert that first */
602:       autoload_func_info spl_alfi;
603:
604:       spl_alfi.func_ptr = spl_func_ptr;
605:       ZVAL_UNDEF(&spl_alfi.obj);
606:       ZVAL_UNDEF(&spl_alfi.closure);
607:       spl_alfi.ce = NULL;
608:       zend_hash_str_add_mem(SPL_G(autoload_functions), "spl_autoload", sizeof("spl_autoload") - 1,
609:           &spl_alfi, sizeof(autoload_func_info));
610:       if (prepend && SPL_G(autoload_functions)->nNumOfElements > 1) {
611:         /* Move the newly created element to the head of the hashtable */
612:         HT_MOVE_TAIL_TO_HEAD(SPL_G(autoload_functions));
613:       }
614:     }
615:
616:     if (UNEXPECTED(alfi.func_ptr == &EG(trampoline))) {
617:       zend_function *copy = emalloc(sizeof(zend_op_array));
618:
619:       memcpy(copy, alfi.func_ptr, sizeof(zend_op_array));
620:       alfi.func_ptr->common.function_name = NULL;
621:       alfi.func_ptr = copy;
622:     }
623:     if (zend_hash_add_mem(SPL_G(autoload_functions), lc_name, &alfi, sizeof(autoload_func_info)) == NULL) {
624:       if (obj_ptr && !(alfi.func_ptr->common.fn_flags & ZEND_ACC_STATIC)) {
625:         Z_DELREF(alfi.obj);
626:       }
627:       if (!Z_ISUNDEF(alfi.closure)) {
628:         Z_DELREF(alfi.closure);
629:       }
630:       if (UNEXPECTED(alfi.func_ptr->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
631:         zend_string_release(alfi.func_ptr->common.function_name);
632:         zend_free_trampoline(alfi.func_ptr);
633:       }
634:     }
635:     if (prepend && SPL_G(autoload_functions)->nNumOfElements > 1) {
636:       /* Move the newly created element to the head of the hashtable */
637:       HT_MOVE_TAIL_TO_HEAD(SPL_G(autoload_functions));
638:     }
639: skip:
640:     zend_string_release(lc_name);
641:   }
642:
643:   if (SPL_G(autoload_functions)) {
644:     EG(autoload_func) = zend_hash_str_find_ptr(EG(function_table), "spl_autoload_call", sizeof("spl_autoload_call") - 1);
645:   } else {
646:     EG(autoload_func) = zend_hash_str_find_ptr(EG(function_table), "spl_autoload", sizeof("spl_autoload") - 1);
647:   }
648:
649:   RETURN_TRUE;
650: } /* }}} */
651:
652: /* {{{ proto bool spl_autoload_unregister(mixed autoload_function)
653:    Unregister given function as __autoload() implementation */
654: PHP_FUNCTION(spl_autoload_unregister)
655: {
656:   zend_string *func_name = NULL;
657:   char *error = NULL;
658:   zend_string *lc_name;
659:   zval *zcallable;
660:   int success = FAILURE;
661:   zend_function *spl_func_ptr;
662:   zend_object *obj_ptr;
663:   zend_fcall_info_cache fcc;
664:
665:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zcallable) == FAILURE) {
666:     return;
667:   }
668:
669:   if (!zend_is_callable_ex(zcallable, NULL, IS_CALLABLE_CHECK_SYNTAX_ONLY, &func_name, &fcc, &error)) {
670:     zend_throw_exception_ex(spl_ce_LogicException, 0, "Unable to unregister invalid function (%s)", error);
671:     if (error) {
672:       efree(error);
673:     }
674:     if (func_name) {
675:       zend_string_release(func_name);
676:     }
677:     RETURN_FALSE;
678:   }
679:   obj_ptr = fcc.object;
680:   if (error) {
681:     efree(error);
682:   }
683:
684:   if (Z_TYPE_P(zcallable) == IS_OBJECT) {
685:     lc_name = zend_string_alloc(ZSTR_LEN(func_name) + sizeof(uint32_t), 0);
686:     zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name), ZSTR_LEN(func_name));
687:     memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(func_name), &Z_OBJ_HANDLE_P(zcallable), sizeof(uint32_t));
688:     ZSTR_VAL(lc_name)[ZSTR_LEN(lc_name)] = '\0';
689:   } else {
690:     /* Skip leading \ */
691:     if (ZSTR_VAL(func_name)[0] == '\\') {
692:       lc_name = zend_string_alloc(ZSTR_LEN(func_name) - 1, 0);
693:       zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name) + 1, ZSTR_LEN(func_name) - 1);
694:     } else {
695:       lc_name = zend_string_tolower(func_name);
696:     }
697:   }
698:   zend_string_release(func_name);
699:
700:   if (SPL_G(autoload_functions)) {
701:     if (ZSTR_LEN(lc_name) == sizeof("spl_autoload_call") - 1 && !strcmp(ZSTR_VAL(lc_name), "spl_autoload_call")) {
702:       /* remove all */
703:       if (!SPL_G(autoload_running)) {
704:         zend_hash_destroy(SPL_G(autoload_functions));
705:         FREE_HASHTABLE(SPL_G(autoload_functions));
706:         SPL_G(autoload_functions) = NULL;
707:         EG(autoload_func) = NULL;
708:       } else {
709:         zend_hash_clean(SPL_G(autoload_functions));
710:       }
711:       success = SUCCESS;
712:     } else {
713:       /* remove specific */
714:       success = zend_hash_del(SPL_G(autoload_functions), lc_name);
715:       if (success != SUCCESS && obj_ptr) {
716:         lc_name = zend_string_extend(lc_name, ZSTR_LEN(lc_name) + sizeof(uint32_t), 0);
717:         memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(lc_name) - sizeof(uint32_t), &obj_ptr->handle, sizeof(uint32_t));
718:         ZSTR_VAL(lc_name)[ZSTR_LEN(lc_name)] = '\0';
719:         success = zend_hash_del(SPL_G(autoload_functions), lc_name);
720:       }
721:     }
722:   } else if (ZSTR_LEN(lc_name) == sizeof("spl_autoload")-1 && !strcmp(ZSTR_VAL(lc_name), "spl_autoload")) {
723:     /* register single spl_autoload() */
724:     spl_func_ptr = zend_hash_str_find_ptr(EG(function_table), "spl_autoload", sizeof("spl_autoload") - 1);
725:
726:     if (EG(autoload_func) == spl_func_ptr) {
727:       success = SUCCESS;
728:       EG(autoload_func) = NULL;
729:     }
730:   }
731:
732:   zend_string_release(lc_name);
733:   RETURN_BOOL(success == SUCCESS);
734: } /* }}} */
735:
736: /* {{{ proto false|array spl_autoload_functions()
737:    Return all registered __autoload() functionns */
738: PHP_FUNCTION(spl_autoload_functions)
739: {
740:   zend_function *fptr;
741:   autoload_func_info *alfi;
742:
743:   if (zend_parse_parameters_none() == FAILURE) {
744:     return;
745:   }
746:
747:   if (!EG(autoload_func)) {
748:     if ((fptr = zend_hash_str_find_ptr(EG(function_table), ZEND_AUTOLOAD_FUNC_NAME, sizeof(ZEND_AUTOLOAD_FUNC_NAME) - 1))) {
749:       array_init(return_value);
750:       add_next_index_string(return_value, ZEND_AUTOLOAD_FUNC_NAME, sizeof(ZEND_AUTOLOAD_FUNC_NAME)-1);
```

```
751:       return;
752:     }
753:     RETURN_FALSE;
754:   }
755:
756:   fptr = zend_hash_str_find_ptr(EG(function_table), "spl_autoload_call", sizeof("spl_autoload_call") - 1);
757:
758:   if (EG(autoload_func) == fptr) {
759:     zend_string *key;
760:     array_init(return_value);
761:     ZEND_HASH_FOREACH_STR_KEY_PTR(SPL_G(autoload_functions), key, alfi) {
762:       if (!Z_ISUNDEF(alfi->closure)) {
763:         Z_ADDREF(alfi->closure);
764:         add_next_index_zval(return_value, &alfi->closure);
765:       } else if (alfi->func_ptr->common.scope) {
766:         zval tmp;
767:
768:         array_init(&tmp);
769:         if (!Z_ISUNDEF(alfi->obj)) {
770:           Z_ADDREF(alfi->obj);
771:           add_next_index_zval(&tmp, &alfi->obj);
772:         } else {
773:           add_next_index_str(&tmp, zend_string_copy(alfi->ce->name));
774:         }
775:         add_next_index_str(&tmp, zend_string_copy(alfi->func_ptr->common.function_name));
776:         add_next_index_zval(return_value, &tmp);
777:       } else {
778:         if (strncmp(ZSTR_VAL(alfi->func_ptr->common.function_name), "__lambda_func", sizeof("__lambda_func") - 1)) {
779:           add_next_index_str(return_value, zend_string_copy(alfi->func_ptr->common.function_name));
780:         } else {
781:           add_next_index_str(return_value, zend_string_copy(key));
782:         }
783:       }
784:     } ZEND_HASH_FOREACH_END();
785:     return;
786:   }
787:
788:   array_init(return_value);
789:   add_next_index_str(return_value, zend_string_copy(EG(autoload_func)->common.function_name));
790: } /* }}} */
791:
792: /* {{{ proto string spl_object_hash(object obj)
793:    Return hash id for given object */
794: PHP_FUNCTION(spl_object_hash)
795: {
796:   zval *obj;
797:
798:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
799:     return;
800:   }
801:
802:   RETURN_NEW_STR(php_spl_object_hash(obj));
803: }
804: /* }}} */
805:
806: /* {{{ proto int spl_object_id(object obj)
807:    Returns the integer object handle for the given object */
808: PHP_FUNCTION(spl_object_id)
809: {
810:   zval *obj;
811:
812:   ZEND_PARSE_PARAMETERS_START(1, 1)
813:     Z_PARAM_OBJECT(obj)
814:   ZEND_PARSE_PARAMETERS_END();
815:
816:   RETURN_LONG((zend_long)Z_OBJ_HANDLE_P(obj));
817: }
818: /* }}} */
819:
820: PHPAPI zend_string *php_spl_object_hash(zval *obj) /* {{{*/
821: {
822:   intptr_t hash_handle, hash_handlers;
823:
824:   if (!SPL_G(hash_mask_init)) {
825:     SPL_G(hash_mask_handle)   = (intptr_t)(php_mt_rand() >> 1);
826:     SPL_G(hash_mask_handlers) = (intptr_t)(php_mt_rand() >> 1);
827:     SPL_G(hash_mask_init) = 1;
828:   }
829:
830:   hash_handle   = SPL_G(hash_mask_handle)^(intptr_t)Z_OBJ_HANDLE_P(obj);
831:   hash_handlers = SPL_G(hash_mask_handlers);
832:
833:   return strpprintf(32, "%016zx%016zx", hash_handle, hash_handlers);
834: }
835: /* }}} */
836:
837: int spl_build_class_list_string(zval *entry, char **list) /* {{{ */
838: {
839:   char *res;
840:
841:   spprintf(&res, 0, "%s, %s", *list, Z_STRVAL_P(entry));
842:   efree(*list);
843:   *list = res;
844:   return ZEND_HASH_APPLY_KEEP;
845: } /* }}} */
846:
847: /* {{{ PHP_MINFO(spl)
848:  */
849: PHP_MINFO_FUNCTION(spl)
850: {
851:   zval list;
852:   char *strg;
853:
854:   php_info_print_table_start();
855:   php_info_print_table_header(2, "SPL support",       "enabled");
856:
857:   array_init(&list);
858:   SPL_LIST_CLASSES(&list, 0, 1, ZEND_ACC_INTERFACE)
859:   strg = estrdup("");
860:   zend_hash_apply_with_argument(Z_ARRVAL_P(&list), (apply_func_arg_t)spl_build_class_list_string, &strg);
861:   zval_dtor(&list);
862:   php_info_print_table_row(2, "Interfaces", strg + 2);
863:   efree(strg);
864:
865:   array_init(&list);
866:   SPL_LIST_CLASSES(&list, 0, -1, ZEND_ACC_INTERFACE)
867:   strg = estrdup("");
868:   zend_hash_apply_with_argument(Z_ARRVAL_P(&list), (apply_func_arg_t)spl_build_class_list_string, &strg);
869:   zval_dtor(&list);
870:   php_info_print_table_row(2, "Classes", strg + 2);
871:   efree(strg);
872:
873:   php_info_print_table_end();
874: }
875: /* }}} */
876:
877: /* {{{ arginfo */
878: ZEND_BEGIN_ARG_INFO_EX(arginfo_iterator_to_array, 0, 0, 1)
879:   ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
880:   ZEND_ARG_INFO(0, use_keys)
881: ZEND_END_ARG_INFO();
882:
883: ZEND_BEGIN_ARG_INFO(arginfo_iterator, 0)
884:   ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
885: ZEND_END_ARG_INFO();
886:
887: ZEND_BEGIN_ARG_INFO_EX(arginfo_iterator_apply, 0, 0, 2)
888:   ZEND_ARG_OBJ_INFO(0, iterator, Traversable, 0)
889:   ZEND_ARG_INFO(0, function)
890:   ZEND_ARG_ARRAY_INFO(0, args, 1)
891: ZEND_END_ARG_INFO();
892:
893: ZEND_BEGIN_ARG_INFO_EX(arginfo_class_parents, 0, 0, 1)
894:   ZEND_ARG_INFO(0, instance)
895:   ZEND_ARG_INFO(0, autoload)
896: ZEND_END_ARG_INFO()
897:
898: ZEND_BEGIN_ARG_INFO_EX(arginfo_class_implements, 0, 0, 1)
899:   ZEND_ARG_INFO(0, what)
900:   ZEND_ARG_INFO(0, autoload)
901: ZEND_END_ARG_INFO()
902:
903: ZEND_BEGIN_ARG_INFO_EX(arginfo_class_uses, 0, 0, 1)
904:   ZEND_ARG_INFO(0, what)
905:   ZEND_ARG_INFO(0, autoload)
906: ZEND_END_ARG_INFO()
907:
908:
909: ZEND_BEGIN_ARG_INFO(arginfo_spl_classes, 0)
910: ZEND_END_ARG_INFO()
911:
912: ZEND_BEGIN_ARG_INFO(arginfo_spl_autoload_functions, 0)
913: ZEND_END_ARG_INFO()
914:
915: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload, 0, 0, 1)
916:   ZEND_ARG_INFO(0, class_name)
917:   ZEND_ARG_INFO(0, file_extensions)
918: ZEND_END_ARG_INFO()
919:
920: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_extensions, 0, 0, 0)
921:   ZEND_ARG_INFO(0, file_extensions)
922: ZEND_END_ARG_INFO()
923:
924: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_call, 0, 0, 1)
925:   ZEND_ARG_INFO(0, class_name)
926: ZEND_END_ARG_INFO()
927:
928: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_register, 0, 0, 0)
929:   ZEND_ARG_INFO(0, autoload_function)
930:   ZEND_ARG_INFO(0, throw)
931:   ZEND_ARG_INFO(0, prepend)
932: ZEND_END_ARG_INFO()
933:
934: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_unregister, 0, 0, 1)
935:   ZEND_ARG_INFO(0, autoload_function)
936: ZEND_END_ARG_INFO()
937:
938: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_object_hash, 0, 0, 1)
939:   ZEND_ARG_INFO(0, obj)
940: ZEND_END_ARG_INFO()
941:
942: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_object_id, 0, 0, 1)
943:   ZEND_ARG_INFO(0, obj)
944: ZEND_END_ARG_INFO()
945: /* }}} */
946:
947: /* {{{ spl_functions
948:  */
949: static const zend_function_entry spl_functions[] = {
950:   PHP_FE(spl_classes,             arginfo_spl_classes)
951:   PHP_FE(spl_autoload,            arginfo_spl_autoload)
952:   PHP_FE(spl_autoload_extensions, arginfo_spl_autoload_extensions)
953:   PHP_FE(spl_autoload_register,   arginfo_spl_autoload_register)
954:   PHP_FE(spl_autoload_unregister, arginfo_spl_autoload_unregister)
955:   PHP_FE(spl_autoload_functions,  arginfo_spl_autoload_functions)
956:   PHP_FE(spl_autoload_call,       arginfo_spl_autoload_call)
957:   PHP_FE(class_parents,           arginfo_class_parents)
958:   PHP_FE(class_implements,        arginfo_class_implements)
959:   PHP_FE(class_uses,              arginfo_class_uses)
960:   PHP_FE(spl_object_hash,         arginfo_spl_object_hash)
961:   PHP_FE(spl_object_id,           arginfo_spl_object_id)
962: #ifdef SPL_ITERATORS_H
963:   PHP_FE(iterator_to_array,       arginfo_iterator_to_array)
964:   PHP_FE(iterator_count,          arginfo_iterator)
965:   PHP_FE(iterator_apply,          arginfo_iterator_apply)
966: #endif /* SPL_ITERATORS_H */
967:   PHP_FE_END
968: };
969: /* }}} */
970:
971: /* {{{ PHP_MINIT_FUNCTION(spl)
972:  */
973: PHP_MINIT_FUNCTION(spl)
974: {
975:   PHP_MINIT(spl_exceptions)(INIT_FUNC_ARGS_PASSTHRU);
976:   PHP_MINIT(spl_iterators)(INIT_FUNC_ARGS_PASSTHRU);
977:   PHP_MINIT(spl_array)(INIT_FUNC_ARGS_PASSTHRU);
978:   PHP_MINIT(spl_directory)(INIT_FUNC_ARGS_PASSTHRU);
979:   PHP_MINIT(spl_dllist)(INIT_FUNC_ARGS_PASSTHRU);
980:   PHP_MINIT(spl_heap)(INIT_FUNC_ARGS_PASSTHRU);
981:   PHP_MINIT(spl_fixedarray)(INIT_FUNC_ARGS_PASSTHRU);
982:   PHP_MINIT(spl_observer)(INIT_FUNC_ARGS_PASSTHRU);
983:
984:   return SUCCESS;
985: }
986: /* }}} */
987:
988: PHP_RINIT_FUNCTION(spl) /* {{{ */
989: {
990:   SPL_G(autoload_extensions) = NULL;
991:   SPL_G(autoload_functions) = NULL;
992:   SPL_G(hash_mask_init) = 0;
993:   return SUCCESS;
994: } /* }}} */
995:
996: PHP_RSHUTDOWN_FUNCTION(spl) /* {{{ */
997: {
998:   if (SPL_G(autoload_extensions)) {
999:     zend_string_release(SPL_G(autoload_extensions));
1000:     SPL_G(autoload_extensions) = NULL;
1001:   }
1002:   if (SPL_G(autoload_functions)) {
1003:     zend_hash_destroy(SPL_G(autoload_functions));
1004:     FREE_HASHTABLE(SPL_G(autoload_functions));
1005:     SPL_G(autoload_functions) = NULL;
1006:   }
1007:   if (SPL_G(hash_mask_init)) {
1008:     SPL_G(hash_mask_init) = 0;
1009:   }
1010:   return SUCCESS;
1011: } /* }}} */
1012:
1013: /* {{{ spl_module_entry
1014:  */
1015: zend_module_entry spl_module_entry = {
1016:   STANDARD_MODULE_HEADER,
1017:   "SPL",
1018:   spl_functions,
1019:   PHP_MINIT(spl),
1020:   NULL,
1021:   PHP_RINIT(spl),
1022:   PHP_RSHUTDOWN(spl),
1023:   PHP_MINFO(spl),
1024:   PHP_SPL_VERSION,
1025:   PHP_MODULE_GLOBALS(spl),
1026:   PHP_GINIT(spl),
1027:   NULL,
1028:   NULL,
1029:   STANDARD_MODULE_PROPERTIES_EX
1030: };
1031: /* }}} */
1032:
1033: /*
1034:  * Local variables:
1035:  * tab-width: 4
1036:  * c-basic-offset: 4
1037:  * End:
1038:  * vim600: fdm=marker
1039:  * vim: noet sw=4 ts=4
1040:  */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "zend_interfaces.h"
#include "zend_exceptions.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_exceptions.h"

PHPAPI zend_class_entry *spl_ce_LogicException;
PHPAPI zend_class_entry *spl_ce_BadFunctionCallException;
PHPAPI zend_class_entry *spl_ce_BadMethodCallException;
PHPAPI zend_class_entry *spl_ce_DomainException;
PHPAPI zend_class_entry *spl_ce_InvalidArgumentException;
PHPAPI zend_class_entry *spl_ce_LengthException;
PHPAPI zend_class_entry *spl_ce_OutOfRangeException;
PHPAPI zend_class_entry *spl_ce_RuntimeException;
PHPAPI zend_class_entry *spl_ce_OutOfBoundsException;
PHPAPI zend_class_entry *spl_ce_OverflowException;
PHPAPI zend_class_entry *spl_ce_RangeException;
PHPAPI zend_class_entry *spl_ce_UnderflowException;
PHPAPI zend_class_entry *spl_ce_UnexpectedValueException;

#define spl_ce_Exception zend_ce_exception

/* {{{ PHP_MINIT_FUNCTION(spl_exceptions) */
PHP_MINIT_FUNCTION(spl_exceptions)
{
    REGISTER_SPL_SUB_CLASS_EX(LogicException,           Exception,        NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(BadFunctionCallException, LogicException,   NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(BadMethodCallException,   BadFunctionCallException,   NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(DomainException,          LogicException,   NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(InvalidArgumentException, LogicException,   NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(LengthException,          LogicException,   NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(OutOfRangeException,      LogicException,   NULL, NULL);

    REGISTER_SPL_SUB_CLASS_EX(RuntimeException,         Exception,        NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(OutOfBoundsException,     RuntimeException, NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(OverflowException,        RuntimeException, NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(RangeException,           RuntimeException, NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(UnderflowException,       RuntimeException, NULL, NULL);
    REGISTER_SPL_SUB_CLASS_EX(UnexpectedValueException, RuntimeException, NULL, NULL);

  return SUCCESS;
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_OBSERVER_H
#define SPL_OBSERVER_H

#include "php.h"
#include "php_spl.h"

extern PHPAPI zend_class_entry *spl_ce_SplObserver;
extern PHPAPI zend_class_entry *spl_ce_SplSubject;
extern PHPAPI zend_class_entry *spl_ce_SplObjectStorage;
extern PHPAPI zend_class_entry *spl_ce_MultipleIterator;

PHP_MINIT_FUNCTION(spl_observer);

#endif /* SPL_OBSERVER_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_EXCEPTIONS_H
#define SPL_EXCEPTIONS_H

#include "php.h"
#include "php_spl.h"

extern PHPAPI zend_class_entry *spl_ce_LogicException;
extern PHPAPI zend_class_entry *spl_ce_BadFunctionCallException;
extern PHPAPI zend_class_entry *spl_ce_BadMethodCallException;
extern PHPAPI zend_class_entry *spl_ce_DomainException;
extern PHPAPI zend_class_entry *spl_ce_InvalidArgumentException;
extern PHPAPI zend_class_entry *spl_ce_LengthException;
extern PHPAPI zend_class_entry *spl_ce_OutOfRangeException;

extern PHPAPI zend_class_entry *spl_ce_RuntimeException;
extern PHPAPI zend_class_entry *spl_ce_OutOfBoundsException;
extern PHPAPI zend_class_entry *spl_ce_OverflowException;
extern PHPAPI zend_class_entry *spl_ce_RangeException;
extern PHPAPI zend_class_entry *spl_ce_UnderflowException;
extern PHPAPI zend_class_entry *spl_ce_UnexpectedValueException;

PHP_MINIT_FUNCTION(spl_exceptions);

#endif /* SPL_EXCEPTIONS_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "php_spl.h"

/* {{{ spl_register_interface */
void spl_register_interface(zend_class_entry ** ppce, char * class_name, const zend_function_entry * functions)
{
	zend_class_entry ce;

	INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), functions);
	*ppce = zend_register_internal_interface(&ce);
}
/* }}} */

/* {{{ spl_register_std_class */
PHPAPI void spl_register_std_class(zend_class_entry ** ppce, char * class_name, void * obj_ctor, const zend_function_entry * function_list)
{
	zend_class_entry ce;

	INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), function_list);
	*ppce = zend_register_internal_class(&ce);

	/* entries changed by initialize */
	if (obj_ctor) {
		(*ppce)->create_object = obj_ctor;
	}
}
/* }}} */

/* {{{ spl_register_sub_class */
PHPAPI void spl_register_sub_class(zend_class_entry ** ppce, zend_class_entry * parent_ce, char * class_name, void *obj_ctor, const zend_function_entry * function_list)
{
	zend_class_entry ce;

	INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), function_list);
	*ppce = zend_register_internal_class_ex(&ce, parent_ce);

	/* entries changed by initialize */
	if (obj_ctor) {
		(*ppce)->create_object = obj_ctor;
	} else {
		(*ppce)->create_object = parent_ce->create_object;
	}
}
/* }}} */

/* {{{ spl_register_property */
void spl_register_property( zend_class_entry * class_entry, char *prop_name, int prop_name_len, int prop_flags)
{
	zend_declare_property_null(class_entry, prop_name, prop_name_len, prop_flags);
}
/* }}} */

/* {{{ spl_add_class_name */
void spl_add_class_name(zval *list, zend_class_entry *pce, int allow, int ce_flags)
{
	if (!allow || (allow > 0 && pce->ce_flags & ce_flags) || (allow < 0 && !(pce->ce_flags & ce_flags))) {
		zval *tmp;

		if ((tmp = zend_hash_find(Z_ARRVAL_P(list), pce->name)) == NULL) {
			zval t;
			ZVAL_STR_COPY(&t, pce->name);
			zend_hash_add(Z_ARRVAL_P(list), pce->name, &t);
		}
	}
}
/* }}} */

/* {{{ spl_add_interfaces */
void spl_add_interfaces(zval *list, zend_class_entry * pce, int allow, int ce_flags)
{
	uint32_t num_interfaces;

	for (num_interfaces = 0; num_interfaces < pce->num_interfaces; num_interfaces++) {
		spl_add_class_name(list, pce->interfaces[num_interfaces], allow, ce_flags);
	}
}
/* }}} */

/* {{{ spl_add_traits */
void spl_add_traits(zval *list, zend_class_entry * pce, int allow, int ce_flags)
{
	uint32_t num_traits;

	for (num_traits = 0; num_traits < pce->num_traits; num_traits++) {
		spl_add_class_name(list, pce->traits[num_traits], allow, ce_flags);
	}
}
/* }}} */


/* {{{ spl_add_classes */
int spl_add_classes(zend_class_entry *pce, zval *list, int sub, int allow, int ce_flags)
{
	if (!pce) {
		return 0;
	}
	spl_add_class_name(list, pce, allow, ce_flags);
	if (sub) {
		spl_add_interfaces(list, pce, allow, ce_flags);
		while (pce->parent) {
			pce = pce->parent;
			spl_add_classes(pce, list, sub, allow, ce_flags);
		}
	}
	return 0;
}
/* }}} */

zend_string * spl_gen_private_prop_name(zend_class_entry *ce, char *prop_name, int prop_len) /* {{{ */
{
	return zend_mangle_property_name(ZSTR_VAL(ce->name), ZSTR_LEN(ce->name), prop_name, prop_len, 0);
}
/* }}} */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

/* $Id$ */

#ifndef SPL_DIRECTORY_H
#define SPL_DIRECTORY_H

#include "php.h"
#include "php_spl.h"

extern PHPAPI zend_class_entry *spl_ce_SplFileInfo;
extern PHPAPI zend_class_entry *spl_ce_DirectoryIterator;
extern PHPAPI zend_class_entry *spl_ce_FilesystemIterator;
extern PHPAPI zend_class_entry *spl_ce_RecursiveDirectoryIterator;
extern PHPAPI zend_class_entry *spl_ce_GlobIterator;
extern PHPAPI zend_class_entry *spl_ce_SplFileObject;
extern PHPAPI zend_class_entry *spl_ce_SplTempFileObject;

PHP_MINIT_FUNCTION(spl_directory);

typedef enum {
  SPL_FS_INFO, /* must be 0 */
  SPL_FS_DIR,
  SPL_FS_FILE
} SPL_FS_OBJ_TYPE;

typedef struct _spl_filesystem_object  spl_filesystem_object;

typedef void (*spl_foreign_dtor_t)(spl_filesystem_object *object);
typedef void (*spl_foreign_clone_t)(spl_filesystem_object *src, spl_filesystem_object *dst);

PHPAPI char* spl_filesystem_object_get_path(spl_filesystem_object *intern, size_t *len);

typedef struct _spl_other_handler {
  spl_foreign_dtor_t    dtor;
  spl_foreign_clone_t   clone;
} spl_other_handler;

/* define an overloaded iterator structure */
typedef struct {
  zend_object_iterator  intern;
  zval                  current;
  void                 *object;
} spl_filesystem_iterator;

struct _spl_filesystem_object {
  void                 *oth;
  const spl_other_handler  *oth_handler;
  char                 *_path;
  size_t               _path_len;
  char                 *orig_path;
  char                 *file_name;
  size_t               file_name_len;
  SPL_FS_OBJ_TYPE      type;
  zend_long             flags;
  zend_class_entry     *file_class;
  zend_class_entry     *info_class;
  union {
    struct {
      php_stream         *dirp;
      php_stream_dirent  entry;
      char               *sub_path;
      size_t             sub_path_len;
      int                index;
      int                is_recursive;
      zend_function      *func_rewind;
      zend_function      *func_next;
      zend_function      *func_valid;
    } dir;
    struct {
      php_stream         *stream;
      php_stream_context *context;
      zval               *zcontext;
      char               *open_mode;
      size_t             open_mode_len;
      zval               current_zval;
      char               *current_line;
      size_t             current_line_len;
      size_t             max_line_len;
      zend_long              current_line_num;
      zval               zresource;
      zend_function      *func_getCurr;
      char               delimiter;
      char               enclosure;
      char               escape;
    } file;
  } u;
  zend_object        std;
};

static inline spl_filesystem_object *spl_filesystem_from_obj(zend_object *obj) /* {{{ */ {
  return (spl_filesystem_object*)((char*)(obj) - XtOffsetOf(spl_filesystem_object, std));
}
/* }}} */

#define Z_SPLFILESYSTEM_P(zv)  spl_filesystem_from_obj(Z_OBJ_P((zv)))

static inline spl_filesystem_iterator* spl_filesystem_object_to_iterator(spl_filesystem_object *obj)
{
  spl_filesystem_iterator    *it;

  it = ecalloc(1, sizeof(spl_filesystem_iterator));
  it->object = (void *)obj;
  zend_iterator_init(&it->intern);
  return it;
}

static inline spl_filesystem_object* spl_filesystem_iterator_to_object(spl_filesystem_iterator *it)
{
  return (spl_filesystem_object*)it->object;
}

#define SPL_FILE_OBJECT_DROP_NEW_LINE       0x00000001 /* drop new lines */
#define SPL_FILE_OBJECT_READ_AHEAD          0x00000002 /* read on rewind/next */
#define SPL_FILE_OBJECT_SKIP_EMPTY          0x00000004 /* skip empty lines */
#define SPL_FILE_OBJECT_READ_CSV            0x00000008 /* read via fgetcsv */
#define SPL_FILE_OBJECT_MASK                0x0000000F /* read via fgetcsv */

#define SPL_FILE_DIR_CURRENT_AS_FILEINFO    0x00000000 /* make RecursiveDirectoryTree::current() return SplFileInfo */
#define SPL_FILE_DIR_CURRENT_AS_SELF        0x00000010 /* make RecursiveDirectoryTree::current() return getSelf() */
#define SPL_FILE_DIR_CURRENT_AS_PATHNAME    0x00000020 /* make RecursiveDirectoryTree::current() return getPathname() */
#define SPL_FILE_DIR_CURRENT_MODE_MASK      0x000000F0 /* mask RecursiveDirectoryTree::current() */
#define SPL_FILE_DIR_CURRENT(intern,mode)  ((intern->flags&SPL_FILE_DIR_CURRENT_MODE_MASK)==mode)

#define SPL_FILE_DIR_KEY_AS_PATHNAME        0x00000000 /* make RecursiveDirectoryTree::key() return getPathname() */
#define SPL_FILE_DIR_KEY_AS_FILENAME        0x00000100 /* make RecursiveDirectoryTree::key() return getFilename() */
#define SPL_FILE_DIR_FOLLOW_SYMLINKS        0x00000200 /* make RecursiveDirectoryTree::hasChildren() follow symlinks */
#define SPL_FILE_DIR_KEY_MODE_MASK          0x00000F00 /* mask RecursiveDirectoryTree::key() */
#define SPL_FILE_DIR_KEY(intern,mode)       ((intern->flags&SPL_FILE_DIR_KEY_MODE_MASK)==mode)

#define SPL_FILE_DIR_SKIPDOTS               0x00001000 /* Tells whether it should skip dots or not */
#define SPL_FILE_DIR_UNIXPATHS              0x00002000 /* Whether to unixify path separators */
#define SPL_FILE_DIR_OTHERS_MASK            0x00003000 /* mask used for get/setFlags */

#endif /* SPL_DIRECTORY_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```

```
/*
  +----------------------------------------------------------------------+
  | PHP Version 7                                                        |
  +----------------------------------------------------------------------+
  | Copyright (c) 1997-2018 The PHP Group                                |
  +----------------------------------------------------------------------+
  | This source file is subject to version 3.01 of the PHP license,      |
  | that is bundled with this package in the file LICENSE, and is        |
  | available through the world-wide-web at the following url:           |
  | http://www.php.net/license/3_01.txt                                  |
  | If you did not receive a copy of the PHP license and are unable to   |
  | obtain it through the world-wide-web, please send a note to          |
  | license@php.net so we can mail you a copy immediately.               |
  +----------------------------------------------------------------------+
  | Author: Antony Dovgal <tony@daylessday.org>                          |
  |         Etienne Kneuss <colder@php.net>                              |
  +----------------------------------------------------------------------+
*/

/* $Id$ */

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "zend_exceptions.h"

#include "php_spl.h"
#include "spl_functions.h"
#include "spl_engine.h"
#include "spl_fixedarray.h"
#include "spl_exceptions.h"
#include "spl_iterators.h"

zend_object_handlers spl_handler_SplFixedArray;
PHPAPI zend_class_entry *spl_ce_SplFixedArray;

#ifdef COMPILE_DL_SPL_FIXEDARRAY
ZEND_GET_MODULE(spl_fixedarray)
#endif

typedef struct _spl_fixedarray { /* {{{ */
  zend_long size;
  zval *elements;
} spl_fixedarray;
/* }}} */

typedef struct _spl_fixedarray_object { /* {{{ */
  spl_fixedarray        array;
  zend_function         *fptr_offset_get;
  zend_function         *fptr_offset_set;
  zend_function         *fptr_offset_has;
  zend_function         *fptr_offset_del;
  zend_function         *fptr_count;
  int                    current;
  int                    flags;
  zend_class_entry      *ce_get_iterator;
  zend_object           std;
} spl_fixedarray_object;
/* }}} */

typedef struct _spl_fixedarray_it { /* {{{ */
  zend_user_iterator     intern;
} spl_fixedarray_it;
/* }}} */

#define SPL_FIXEDARRAY_OVERLOADED_REWIND  0x0001
#define SPL_FIXEDARRAY_OVERLOADED_VALID   0x0002
#define SPL_FIXEDARRAY_OVERLOADED_KEY     0x0004
#define SPL_FIXEDARRAY_OVERLOADED_CURRENT 0x0008
#define SPL_FIXEDARRAY_OVERLOADED_NEXT    0x0010

static inline spl_fixedarray_object *spl_fixed_array_from_obj(zend_object *obj) /* {{{ */ {
  return (spl_fixedarray_object*)((char*)(obj) - XtOffsetOf(spl_fixedarray_object, std));
}
/* }}} */

#define Z_SPLFIXEDARRAY_P(zv)  spl_fixed_array_from_obj(Z_OBJ_P((zv)))

static void spl_fixedarray_init(spl_fixedarray *array, zend_long size) /* {{{ */
{
  if (size > 0) {
    array->size = 0; /* reset size in case ecalloc() fails */
    array->elements = ecalloc(size, sizeof(zval));
    array->size = size;
  } else {
    array->elements = NULL;
    array->size = 0;
  }
}
/* }}} */

static void spl_fixedarray_resize(spl_fixedarray *array, zend_long size) /* {{{ */
{
  if (size == array->size) {
    /* nothing to do */
    return;
  }

  /* first initialization */
  if (array->size == 0) {
    spl_fixedarray_init(array, size);
    return;
  }

  /* clearing the array */
  if (size == 0) {
    zend_long i;

    for (i = 0; i < array->size; i++) {
      zval_ptr_dtor(&(array->elements[i]));
    }

    if (array->elements) {
      efree(array->elements);
      array->elements = NULL;
    }
  } else if (size > array->size) {
    array->elements = safe_erealloc(array->elements, size, sizeof(zval), 0);
    memset(array->elements + array->size, '\0', sizeof(zval) * (size - array->size));
  } else { /* size < array->size */
    zend_long i;

    for (i = size; i < array->size; i++) {
      zval_ptr_dtor(&(array->elements[i]));
    }
    array->elements = erealloc(array->elements, sizeof(zval) * size);
  }

  array->size = size;
}
/* }}} */

static void spl_fixedarray_copy(spl_fixedarray *to, spl_fixedarray *from) /* {{{ */
{
  int i;
  for (i = 0; i < from->size; i++) {
    ZVAL_COPY(&to->elements[i], &from->elements[i]);
  }
}
/* }}} */

static HashTable* spl_fixedarray_object_get_gc(zval *obj, zval **table, int *n) /* {{{ */
{
  spl_fixedarray_object *intern  = Z_SPLFIXEDARRAY_P(obj);
  HashTable *ht = zend_std_get_properties(obj);

  *table = intern->array.elements;
  *n = (int)intern->array.size;

  return ht;
}
/* }}} */

static HashTable* spl_fixedarray_object_get_properties(zval *obj) /* {{{ */
{
  spl_fixedarray_object *intern  = Z_SPLFIXEDARRAY_P(obj);
  HashTable *ht = zend_std_get_properties(obj);
  zend_long  i = 0;

  if (intern->array.size > 0) {
    zend_long j = zend_hash_num_elements(ht);

    for (i = 0; i < intern->array.size; i++) {
      if (!Z_ISUNDEF(intern->array.elements[i])) {
        zend_hash_index_update(ht, i, &intern->array.elements[i]);
        Z_TRY_ADDREF(intern->array.elements[i]);
      } else {
        zend_hash_index_update(ht, i, &EG(uninitialized_zval));
      }
    }
    if (j > intern->array.size) {
      for (i = intern->array.size; i < j; ++i) {
        zend_hash_index_del(ht, i);
      }
    }
  }

  return ht;
}
/* }}} */

static void spl_fixedarray_object_free_storage(zend_object *object) /* {{{ */
{
  spl_fixedarray_object *intern = spl_fixed_array_from_obj(object);

  zend_long i;

  if (intern->array.size > 0) {
    for (i = 0; i < intern->array.size; i++) {
      zval_ptr_dtor(&(intern->array.elements[i]));
    }

    if (intern->array.elements) {
      efree(intern->array.elements);
    }
  }

  zend_object_std_dtor(&intern->std);
}
/* }}} */

zend_object_iterator *spl_fixedarray_get_iterator(zend_class_entry *ce, zval *object, int by_ref);

static zend_object *spl_fixedarray_object_new_ex(zend_class_entry *class_type, zval *orig, int clone_orig) /* {{{ */
{
  spl_fixedarray_object *intern;
  zend_class_entry      *parent = class_type;
  int                    inherited = 0;

  intern = zend_object_alloc(sizeof(spl_fixedarray_object), parent);

  zend_object_std_init(&intern->std, class_type);
  object_properties_init(&intern->std, class_type);

  intern->current = 0;
  intern->flags = 0;

  if (orig && clone_orig) {
    spl_fixedarray_object *other = Z_SPLFIXEDARRAY_P(orig);
    intern->ce_get_iterator = other->ce_get_iterator;
    spl_fixedarray_init(&intern->array, other->array.size);
    spl_fixedarray_copy(&intern->array, &other->array);
  }

  while (parent) {
    if (parent == spl_ce_SplFixedArray) {
      intern->std.handlers = &spl_handler_SplFixedArray;
      class_type->get_iterator = spl_fixedarray_get_iterator;
      break;
    }

    parent = parent->parent;
    inherited = 1;
  }

  if (!parent) { /* this must never happen */
    php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error, Class is not child of SplFixedArray");
  }

  if (!class_type->iterator_funcs.zf_current) {
    class_type->iterator_funcs.zf_rewind = zend_hash_str_find_ptr(&class_type->function_table, "rewind", sizeof("rewind") - 1);
    class_type->iterator_funcs.zf_valid = zend_hash_str_find_ptr(&class_type->function_table, "valid", sizeof("valid") - 1);
    class_type->iterator_funcs.zf_key = zend_hash_str_find_ptr(&class_type->function_table, "key", sizeof("key") - 1);
    class_type->iterator_funcs.zf_current = zend_hash_str_find_ptr(&class_type->function_table, "current", sizeof("current") - 1);
    class_type->iterator_funcs.zf_next = zend_hash_str_find_ptr(&class_type->function_table, "next", sizeof("next") - 1);
  }
  if (inherited) {
    if (class_type->iterator_funcs.zf_rewind->common.scope  != parent) {
      intern->flags |= SPL_FIXEDARRAY_OVERLOADED_REWIND;
    }
    if (class_type->iterator_funcs.zf_valid->common.scope   != parent) {
      intern->flags |= SPL_FIXEDARRAY_OVERLOADED_VALID;
    }
    if (class_type->iterator_funcs.zf_key->common.scope     != parent) {
      intern->flags |= SPL_FIXEDARRAY_OVERLOADED_KEY;
    }
    if (class_type->iterator_funcs.zf_current->common.scope != parent) {
      intern->flags |= SPL_FIXEDARRAY_OVERLOADED_CURRENT;
    }
    if (class_type->iterator_funcs.zf_next->common.scope    != parent) {
      intern->flags |= SPL_FIXEDARRAY_OVERLOADED_NEXT;
    }

    intern->fptr_offset_get = zend_hash_str_find_ptr(&class_type->function_table, "offsetget", sizeof("offsetget") - 1);
    if (intern->fptr_offset_get->common.scope == parent) {
      intern->fptr_offset_get = NULL;
    }
    intern->fptr_offset_set = zend_hash_str_find_ptr(&class_type->function_table, "offsetset", sizeof("offsetset") - 1);
    if (intern->fptr_offset_set->common.scope == parent) {
      intern->fptr_offset_set = NULL;
    }
    intern->fptr_offset_has = zend_hash_str_find_ptr(&class_type->function_table, "offsetexists", sizeof("offsetexists") - 1);
    if (intern->fptr_offset_has->common.scope == parent) {
      intern->fptr_offset_has = NULL;
    }
    intern->fptr_offset_del = zend_hash_str_find_ptr(&class_type->function_table, "offsetunset", sizeof("offsetunset") - 1);
    if (intern->fptr_offset_del->common.scope == parent) {
      intern->fptr_offset_del = NULL;
    }
    intern->fptr_count = zend_hash_str_find_ptr(&class_type->function_table, "count", sizeof("count") - 1);
    if (intern->fptr_count->common.scope == parent) {
      intern->fptr_count = NULL;
    }
  }

  return &intern->std;
}
/* }}} */

static zend_object *spl_fixedarray_new(zend_class_entry *class_type) /* {{{ */
{
  return spl_fixedarray_object_new_ex(class_type, NULL, 0);
}
/* }}} */

static zend_object *spl_fixedarray_object_clone(zval *zobject) /* {{{ */
{
  zend_object *old_object;
  zend_object *new_object;

  old_object  = Z_OBJ_P(zobject);
  new_object = spl_fixedarray_object_new_ex(old_object->ce, zobject, 1);

  zend_objects_clone_members(new_object, old_object);

  return new_object;
}
/* }}} */

static inline zval *spl_fixedarray_object_read_dimension_helper(spl_fixedarray_object *intern, zval *offset) /* {{{ */
{
  zend_long index;

  /* we have to return NULL on error here to avoid memleak because of
   * ZE duplicating uninitialized_zval_ptr */
  if (!offset) {
    zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
    return NULL;
  }

  if (Z_TYPE_P(offset) != IS_LONG) {
    index = spl_offset_convert_to_long(offset);
  } else {
    index = Z_LVAL_P(offset);
  }

  if (index < 0 || index >= intern->array.size) {
    zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
    return NULL;
  } else if (Z_ISUNDEF(intern->array.elements[index])) {
    return NULL;
  } else {
    return &intern->array.elements[index];
  }
}
/* }}} */

static zval *spl_fixedarray_object_read_dimension(zval *object, zval *offset, int type, zval *rv) /* {{{ */
{
  spl_fixedarray_object *intern;

  intern = Z_SPLFIXEDARRAY_P(object);

  if (type == BP_VAR_IS && intern->fptr_offset_has) {
    SEPARATE_ARG_IF_REF(offset);
    zend_call_method_with_1_params(object, intern->std.ce, &intern->fptr_offset_has, "offsetexists", rv, offset);
    if (UNEXPECTED(Z_ISUNDEF_P(rv))) {
      zval_ptr_dtor(offset);
      return NULL;
    }
    if (!i_zend_is_true(rv)) {
      zval_ptr_dtor(offset);
      zval_ptr_dtor(rv);
      return &EG(uninitialized_zval);
    }
    zval_ptr_dtor(rv);
  }

  if (intern->fptr_offset_get) {
    zval tmp;
    if (!offset) {
      ZVAL_NULL(&tmp);
      offset = &tmp;
    } else {
      SEPARATE_ARG_IF_REF(offset);
    }
    zend_call_method_with_1_params(object, intern->std.ce, &intern->fptr_offset_get, "offsetGet", rv, offset);
    zval_ptr_dtor(offset);
    if (!Z_ISUNDEF_P(rv)) {
      return rv;
    }
    return &EG(uninitialized_zval);
  }
```

```
377:
378:   return spl_fixedarray_object_read_dimension_helper(intern, offset);
379: }
380: /* }}} */
381:
382: static inline void spl_fixedarray_object_write_dimension_helper(spl_fixedarray_object *intern, zval *offset, zval *value) /* {{{ */
383: {
384:   zend_long index;
385:
386:   if (!offset) {
387:     /* '$array[] = value' syntax is not supported */
388:     zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
389:     return;
390:   }
391:
392:   if (Z_TYPE_P(offset) != IS_LONG) {
393:     index = spl_offset_convert_to_long(offset);
394:   } else {
395:     index = Z_LVAL_P(offset);
396:   }
397:
398:   if (index < 0 || index >= intern->array.size) {
399:     zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
400:     return;
401:   } else {
402:     if (!Z_ISUNDEF(intern->array.elements[index])) {
403:       zval_ptr_dtor(&(intern->array.elements[index]));
404:     }
405:     ZVAL_DEREF(value);
406:     ZVAL_COPY(&intern->array.elements[index], value);
407:   }
408: }
409: /* }}} */
410:
411: static void spl_fixedarray_object_write_dimension(zval *object, zval *offset, zval *value) /* {{{ */
412: {
413:   spl_fixedarray_object *intern;
414:   zval tmp;
415:
416:   intern = Z_SPLFIXEDARRAY_P(object);
417:
418:   if (intern->fptr_offset_set) {
419:     if (!offset) {
420:       ZVAL_NULL(&tmp);
421:       offset = &tmp;
422:     } else {
423:       SEPARATE_ARG_IF_REF(offset);
424:     }
425:     SEPARATE_ARG_IF_REF(value);
426:     zend_call_method_with_2_params(object, intern->std.ce, &intern->fptr_offset_set, "offsetSet", NULL, offset, value);
427:     zval_ptr_dtor(value);
428:     zval_ptr_dtor(offset);
429:     return;
430:   }
431:   spl_fixedarray_object_write_dimension_helper(intern, offset, value);
432: }
433: /* }}} */
434:
435: static inline void spl_fixedarray_object_unset_dimension_helper(spl_fixedarray_object *intern, zval *offset) /* {{{ */
436: {
437:   zend_long index;
438:
439:   if (Z_TYPE_P(offset) != IS_LONG) {
440:     index = spl_offset_convert_to_long(offset);
441:   } else {
442:     index = Z_LVAL_P(offset);
443:   }
444:
445:   if (index < 0 || index >= intern->array.size) {
446:     zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
447:     return;
448:   } else {
449:     zval_ptr_dtor(&(intern->array.elements[index]));
450:     ZVAL_UNDEF(&intern->array.elements[index]);
451:   }
452: }
453: /* }}} */
454:
455: static void spl_fixedarray_object_unset_dimension(zval *object, zval *offset) /* {{{ */
456: {
457:   spl_fixedarray_object *intern;
458:
459:   intern = Z_SPLFIXEDARRAY_P(object);
460:
461:   if (intern->fptr_offset_del) {
462:     SEPARATE_ARG_IF_REF(offset);
463:     zend_call_method_with_1_params(object, intern->std.ce, &intern->fptr_offset_del, "offsetUnset", NULL, offset);
464:     zval_ptr_dtor(offset);
465:     return;
466:   }
467:
468:   spl_fixedarray_object_unset_dimension_helper(intern, offset);
469:
470: }
471: /* }}} */
472:
473: static inline int spl_fixedarray_object_has_dimension_helper(spl_fixedarray_object *intern, zval *offset, int check_empty) /* {{{ */
474: {
475:   zend_long index;
476:   int retval;
477:
478:   if (Z_TYPE_P(offset) != IS_LONG) {
479:     index = spl_offset_convert_to_long(offset);
480:   } else {
481:     index = Z_LVAL_P(offset);
482:   }
483:
484:   if (index < 0 || index >= intern->array.size) {
485:     retval = 0;
486:   } else {
487:     if (Z_ISUNDEF(intern->array.elements[index])) {
488:       retval = 0;
489:     } else if (check_empty) {
490:       if (zend_is_true(&intern->array.elements[index])) {
491:         retval = 1;
492:       } else {
493:         retval = 0;
494:       }
495:     } else { /* != NULL and !check_empty */
496:       retval = 1;
497:     }
498:   }
499:
500:   return retval;
501: }
502: /* }}} */
503:
504: static int spl_fixedarray_object_has_dimension(zval *object, zval *offset, int check_empty) /* {{{ */
505: {
506:   spl_fixedarray_object *intern;
507:
508:   intern = Z_SPLFIXEDARRAY_P(object);
509:
510:   if (intern->fptr_offset_has) {
511:     zval rv;
512:     SEPARATE_ARG_IF_REF(offset);
513:     zend_call_method_with_1_params(object, intern->std.ce, &intern->fptr_offset_has, "offsetExists", &rv, offset);
514:     zval_ptr_dtor(offset);
515:     if (!Z_ISUNDEF(rv)) {
516:       zend_bool result = zend_is_true(&rv);
517:       zval_ptr_dtor(&rv);
518:       return result;
519:     }
520:     return 0;
521:   }
522:
523:   return spl_fixedarray_object_has_dimension_helper(intern, offset, check_empty);
524: }
525: /* }}} */
526:
527: static int spl_fixedarray_object_count_elements(zval *object, zend_long *count) /* {{{ */
528: {
529:   spl_fixedarray_object *intern;
530:
531:   intern = Z_SPLFIXEDARRAY_P(object);
532:   if (intern->fptr_count) {
533:     zval rv;
534:     zend_call_method_with_0_params(object, intern->std.ce, &intern->fptr_count, "count", &rv);
535:     if (!Z_ISUNDEF(rv)) {
536:       *count = zval_get_long(&rv);
537:       zval_ptr_dtor(&rv);
538:     } else {
539:       *count = 0;
540:     }
541:   } else {
542:     *count = intern->array.size;
543:   }
544:   return SUCCESS;
545: }
546: /* }}} */
547:
548: /* {{{ proto void SplFixedArray::__construct([int size])
549: */
550: SPL_METHOD(SplFixedArray, __construct)
551: {
552:   zval *object = getThis();
553:   spl_fixedarray_object *intern;
554:   zend_long size = 0;
555:
556:   if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "|l", &size) == FAILURE) {
557:     return;
558:   }
559:
560:   if (size < 0) {
561:     zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "array size cannot be less than zero");
562:     return;
563:   }
564:
565:
566:   intern = Z_SPLFIXEDARRAY_P(object);
567:
568:   if (intern->array.size > 0) {
569:     /* called __construct() twice, bail out */
570:     return;
571:   }
572:
573:   spl_fixedarray_init(&intern->array, size);
574: }
575: /* }}} */
576:
577: /* {{{ proto void SplFixedArray::__wakeup()
578: */
579: SPL_METHOD(SplFixedArray, __wakeup)
580: {
581:   spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());
582:   HashTable *intern_ht = zend_std_get_properties(getThis());
583:   zval *data;
584:
585:   if (zend_parse_parameters_none() == FAILURE) {
586:     return;
587:   }
588:
589:   if (intern->array.size == 0) {
590:     int index = 0;
591:     int size = zend_hash_num_elements(intern_ht);
592:
593:     spl_fixedarray_init(&intern->array, size);
594:
595:     ZEND_HASH_FOREACH_VAL(intern_ht, data) {
596:       ZVAL_COPY(&intern->array.elements[index], data);
597:       index++;
598:     } ZEND_HASH_FOREACH_END();
599:
600:     /* Remove the unserialised properties, since we now have the elements
601:      * within the spl_fixedarray_object structure. */
602:     zend_hash_clean(intern_ht);
603:   }
604: }
605: /* }}} */
606:
607: /* {{{ proto int SplFixedArray::count(void)
608: */
609: SPL_METHOD(SplFixedArray, count)
610: {
611:   zval *object = getThis();
612:   spl_fixedarray_object *intern;
613:
614:   if (zend_parse_parameters_none() == FAILURE) {
615:     return;
616:   }
617:
618:   intern = Z_SPLFIXEDARRAY_P(object);
619:   RETURN_LONG(intern->array.size);
620: }
621: /* }}} */
622:
623: /* {{{ proto object SplFixedArray::toArray()
624: */
625: SPL_METHOD(SplFixedArray, toArray)
626: {
627:   spl_fixedarray_object *intern;
628:
629:   if (zend_parse_parameters_none() == FAILURE) {
630:     return;
631:   }
632:
633:   intern = Z_SPLFIXEDARRAY_P(getThis());
634:
635:   if (intern->array.size > 0) {
636:     int i = 0;
637:
638:     array_init(return_value);
639:     for (; i < intern->array.size; i++) {
640:       if (!Z_ISUNDEF(intern->array.elements[i])) {
641:         zend_hash_index_update(Z_ARRVAL_P(return_value), i, &intern->array.elements[i]);
642:         Z_TRY_ADDREF(intern->array.elements[i]);
643:       } else {
644:         zend_hash_index_update(Z_ARRVAL_P(return_value), i, &EG(uninitialized_zval));
645:       }
646:     }
647:   } else {
648:     ZVAL_EMPTY_ARRAY(return_value);
649:   }
650: }
651: /* }}} */
652:
653: /* {{{ proto object SplFixedArray::fromArray(array data[, bool save_indexes])
654: */
655: SPL_METHOD(SplFixedArray, fromArray)
656: {
657:   zval *data;
658:   spl_fixedarray array;
659:   spl_fixedarray_object *intern;
660:   int num;
661:   zend_bool save_indexes = 1;
662:
663:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "a|b", &data, &save_indexes) == FAILURE) {
664:     return;
665:   }
666:
667:   num = zend_hash_num_elements(Z_ARRVAL_P(data));
668:
669:   if (num > 0 && save_indexes) {
670:     zval *element;
671:     zend_string *str_index;
672:     zend_ulong num_index, max_index = 0;
673:     zend_long tmp;
674:
675:     ZEND_HASH_FOREACH_KEY(Z_ARRVAL_P(data), num_index, str_index) {
676:       if (str_index != NULL || (zend_long)num_index < 0) {
677:         zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "array must contain only positive integer keys");
678:         return;
679:       }
680:
681:       if (num_index > max_index) {
682:         max_index = num_index;
683:       }
684:     } ZEND_HASH_FOREACH_END();
685:
686:     tmp = max_index + 1;
687:     if (tmp <= 0) {
688:       zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "integer overflow detected");
689:       return;
690:     }
691:     spl_fixedarray_init(&array, tmp);
692:
693:     ZEND_HASH_FOREACH_KEY_VAL(Z_ARRVAL_P(data), num_index, str_index, element) {
694:       ZVAL_DEREF(element);
695:       ZVAL_COPY(&array.elements[num_index], element);
696:     } ZEND_HASH_FOREACH_END();
697:
698:   } else if (num > 0 && !save_indexes) {
699:     zval *element;
700:     zend_long i = 0;
701:
702:     spl_fixedarray_init(&array, num);
703:
704:     ZEND_HASH_FOREACH_VAL(Z_ARRVAL_P(data), element) {
705:       ZVAL_DEREF(element);
706:       ZVAL_COPY(&array.elements[i], element);
707:       i++;
708:     } ZEND_HASH_FOREACH_END();
709:   } else {
710:     spl_fixedarray_init(&array, 0);
711:   }
712:
713:   object_init_ex(return_value, spl_ce_SplFixedArray);
714:
715:   intern = Z_SPLFIXEDARRAY_P(return_value);
716:   intern->array = array;
717: }
718: /* }}} */
719:
720: /* {{{ proto int SplFixedArray::getSize(void)
721: */
722: SPL_METHOD(SplFixedArray, getSize)
723: {
724:   zval *object = getThis();
725:   spl_fixedarray_object *intern;
726:
727:   if (zend_parse_parameters_none() == FAILURE) {
728:     return;
729:   }
730:
731:   intern = Z_SPLFIXEDARRAY_P(object);
732:   RETURN_LONG(intern->array.size);
733: }
734: /* }}} */
735:
736: /* {{{ proto bool SplFixedArray::setSize(int size)
737: */
738: SPL_METHOD(SplFixedArray, setSize)
739: {
740:   zval *object = getThis();
741:   spl_fixedarray_object *intern;
742:   zend_long size;
743:
744:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &size) == FAILURE) {
745:     return;
746:   }
747:
748:   if (size < 0) {
749:     zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "array size cannot be less than zero");
750:     return;
751:   }
752:
```

```
753:   intern = Z_SPLFIXEDARRAY_P(object);
754:
755:   spl_fixedarray_resize(&intern->array, size);
756:   RETURN_TRUE;
757: }
758: /* }}} */
759:
760: /* {{{ proto bool SplFixedArray::offsetExists(mixed $index)
761:   Returns whether the requested $index exists. */
762: SPL_METHOD(SplFixedArray, offsetExists)
763: {
764:   zval               *zindex;
765:   spl_fixedarray_object  *intern;
766:
767:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
768:     return;
769:   }
770:
771:   intern = Z_SPLFIXEDARRAY_P(getThis());
772:
773:   RETURN_BOOL(spl_fixedarray_object_has_dimension_helper(intern, zindex, 0));
774: } /* }}} */
775:
776: /* {{{ proto mixed SplFixedArray::offsetGet(mixed $index)
777:   Returns the value at the specified $index. */
778: SPL_METHOD(SplFixedArray, offsetGet)
779: {
780:   zval *zindex, *value;
781:   spl_fixedarray_object  *intern;
782:
783:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
784:     return;
785:   }
786:
787:   intern = Z_SPLFIXEDARRAY_P(getThis());
788:   value = spl_fixedarray_object_read_dimension_helper(intern, zindex);
789:
790:   if (value) {
791:     ZVAL_DEREF(value);
792:     ZVAL_COPY(return_value, value);
793:   } else {
794:     RETURN_NULL();
795:   }
796: } /* }}} */
797:
798: /* {{{ proto void SplFixedArray::offsetSet(mixed $index, mixed $newval)
799:   Sets the value at the specified $index to $newval. */
800: SPL_METHOD(SplFixedArray, offsetSet)
801: {
802:   zval                *zindex, *value;
803:   spl_fixedarray_object  *intern;
804:
805:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "zz", &zindex, &value) == FAILURE) {
806:     return;
807:   }
808:
809:   intern = Z_SPLFIXEDARRAY_P(getThis());
810:   spl_fixedarray_object_write_dimension_helper(intern, zindex, value);
811:
812: } /* }}} */
813:
814: /* {{{ proto void SplFixedArray::offsetUnset(mixed $index)
815:   Unsets the value at the specified $index. */
816: SPL_METHOD(SplFixedArray, offsetUnset)
817: {
818:   zval                *zindex;
819:   spl_fixedarray_object  *intern;
820:
821:   if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
822:     return;
823:   }
824:
825:   intern = Z_SPLFIXEDARRAY_P(getThis());
826:   spl_fixedarray_object_unset_dimension_helper(intern, zindex);
827:
828: } /* }}} */
829:
830: static void spl_fixedarray_it_dtor(zend_object_iterator *iter) /* {{{ */
831: {
832:   spl_fixedarray_it  *iterator = (spl_fixedarray_it *)iter;
833:
834:   zend_user_it_invalidate_current(iter);
835:   zval_ptr_dtor(&iterator->intern.it.data);
836: }
837: /* }}} */
838:
839: static void spl_fixedarray_it_rewind(zend_object_iterator *iter) /* {{{ */
840: {
841:   spl_fixedarray_object *object = Z_SPLFIXEDARRAY_P(&iter->data);
842:
843:   if (object->flags & SPL_FIXEDARRAY_OVERLOADED_REWIND) {
844:     zend_user_it_rewind(iter);
845:   } else {
846:     object->current = 0;
847:   }
848: }
849: /* }}} */
850:
851: static int spl_fixedarray_it_valid(zend_object_iterator *iter) /* {{{ */
852: {
853:   spl_fixedarray_object *object = Z_SPLFIXEDARRAY_P(&iter->data);
854:
855:   if (object->flags & SPL_FIXEDARRAY_OVERLOADED_VALID) {
856:     return zend_user_it_valid(iter);
857:   }
858:
859:   if (object->current >= 0 && object->current < object->array.size) {
860:     return SUCCESS;
861:   }
862:
863:   return FAILURE;
864: }
865: /* }}} */
866:
867: static zval *spl_fixedarray_it_get_current_data(zend_object_iterator *iter) /* {{{ */
868: {
869:   zval zindex;
870:   spl_fixedarray_object *object = Z_SPLFIXEDARRAY_P(&iter->data);
871:
872:   if (object->flags & SPL_FIXEDARRAY_OVERLOADED_CURRENT) {
873:     return zend_user_it_get_current_data(iter);
874:   } else {
875:     zval *data;
876:
877:     ZVAL_LONG(&zindex, object->current);
878:
879:     data = spl_fixedarray_object_read_dimension_helper(object, &zindex);
880:     zval_ptr_dtor(&zindex);
881:
882:     if (data == NULL) {
883:       data = &EG(uninitialized_zval);
884:     }
885:     return data;
886:   }
887: }
888: /* }}} */
889:
890: static void spl_fixedarray_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
891: {
892:   spl_fixedarray_object *object = Z_SPLFIXEDARRAY_P(&iter->data);
893:
894:   if (object->flags & SPL_FIXEDARRAY_OVERLOADED_KEY) {
895:     zend_user_it_get_current_key(iter, key);
896:   } else {
897:     ZVAL_LONG(key, object->current);
898:   }
899: }
900: /* }}} */
901:
902: static void spl_fixedarray_it_move_forward(zend_object_iterator *iter) /* {{{ */
903: {
904:   spl_fixedarray_object *object = Z_SPLFIXEDARRAY_P(&iter->data);
905:
906:   if (object->flags & SPL_FIXEDARRAY_OVERLOADED_NEXT) {
907:     zend_user_it_move_forward(iter);
908:   } else {
909:     zend_user_it_invalidate_current(iter);
910:     object->current++;
911:   }
912: }
913: /* }}} */
914:
915: /* {{{ proto int SplFixedArray::key()
916:   Return current array key */
917: SPL_METHOD(SplFixedArray, key)
918: {
919:   spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());
920:
921:   if (zend_parse_parameters_none() == FAILURE) {
922:     return;
923:   }
924:
925:   RETURN_LONG(intern->current);
926: }
927: /* }}} */
928:
929: /* {{{ proto void SplFixedArray::next()
930:   Move to next entry */
931: SPL_METHOD(SplFixedArray, next)
932: {
933:   spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());
934:
935:   if (zend_parse_parameters_none() == FAILURE) {
936:     return;
937:   }
938:
939:   intern->current++;
940: }
941: /* }}} */
942:
943: /* {{{ proto bool SplFixedArray::valid()
944:   Check whether the datastructure contains more entries */
945: SPL_METHOD(SplFixedArray, valid)
946: {
947:   spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());
948:
949:   if (zend_parse_parameters_none() == FAILURE) {
950:     return;
951:   }
952:
953:   RETURN_BOOL(intern->current >= 0 && intern->current < intern->array.size);
954: }
955: /* }}} */
956:
957: /* {{{ proto void SplFixedArray::rewind()
958:   Rewind the datastructure back to the start */
959: SPL_METHOD(SplFixedArray, rewind)
960: {
961:   spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());
962:
963:   if (zend_parse_parameters_none() == FAILURE) {
964:     return;
965:   }
966:
967:   intern->current = 0;
968: }
969: /* }}} */
970:
971: /* {{{ proto mixed|NULL SplFixedArray::current()
972:   Return current datastructure entry */
973: SPL_METHOD(SplFixedArray, current)
974: {
975:   zval zindex, *value;
976:   spl_fixedarray_object *intern  = Z_SPLFIXEDARRAY_P(getThis());
977:
978:   if (zend_parse_parameters_none() == FAILURE) {
979:     return;
980:   }
981:
982:   ZVAL_LONG(&zindex, intern->current);
983:
984:   value = spl_fixedarray_object_read_dimension_helper(intern, &zindex);
985:
986:   zval_ptr_dtor(&zindex);
987:
988:   if (value) {
989:     ZVAL_DEREF(value);
990:     ZVAL_COPY(return_value, value);
991:   } else {
992:     RETURN_NULL();
993:   }
994: }
995: /* }}} */
996:
997: /* iterator handler table */
998: static const zend_object_iterator_funcs spl_fixedarray_it_funcs = {
999:   spl_fixedarray_it_dtor,
1000:   spl_fixedarray_it_valid,
1001:   spl_fixedarray_it_get_current_data,
1002:   spl_fixedarray_it_get_current_key,
1003:   spl_fixedarray_it_move_forward,
1004:   spl_fixedarray_it_rewind,
1005:   NULL
1006: };
1007:
1008: zend_object_iterator *spl_fixedarray_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1009: {
1010:   spl_fixedarray_it *iterator;
1011:
1012:   if (by_ref) {
1013:     zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1014:     return NULL;
1015:   }
1016:
1017:   iterator = emalloc(sizeof(spl_fixedarray_it));
1018:
1019:   zend_iterator_init((zend_object_iterator*)iterator);
1020:
1021:   ZVAL_COPY(&iterator->intern.it.data, object);
1022:   iterator->intern.it.funcs = &spl_fixedarray_it_funcs;
1023:   iterator->intern.ce = ce;
1024:   ZVAL_UNDEF(&iterator->intern.value);
1025:
1026:   return &iterator->intern.it;
1027: }
1028: /* }}} */
1029:
1030: ZEND_BEGIN_ARG_INFO_EX(arginfo_splfixedarray_construct, 0, 0, 0)
1031:   ZEND_ARG_INFO(0, size)
1032: ZEND_END_ARG_INFO()
1033:
1034: ZEND_BEGIN_ARG_INFO_EX(arginfo_fixedarray_offsetGet, 0, 0, 1)
1035:   ZEND_ARG_INFO(0, index)
1036: ZEND_END_ARG_INFO()
1037:
1038: ZEND_BEGIN_ARG_INFO_EX(arginfo_fixedarray_offsetSet, 0, 0, 2)
1039:   ZEND_ARG_INFO(0, index)
1040:   ZEND_ARG_INFO(0, newval)
1041: ZEND_END_ARG_INFO()
1042:
1043: ZEND_BEGIN_ARG_INFO(arginfo_fixedarray_setSize, 0)
1044:   ZEND_ARG_INFO(0, value)
1045: ZEND_END_ARG_INFO()
1046:
1047: ZEND_BEGIN_ARG_INFO_EX(arginfo_fixedarray_fromArray, 0, 0, 1)
1048:   ZEND_ARG_INFO(0, data)
1049:   ZEND_ARG_INFO(0, save_indexes)
1050: ZEND_END_ARG_INFO()
1051:
1052: ZEND_BEGIN_ARG_INFO(arginfo_splfixedarray_void, 0)
1053: ZEND_END_ARG_INFO()
1054:
1055: static const zend_function_entry spl_funcs_SplFixedArray[] = { /* {{{ */
1056:   SPL_ME(SplFixedArray, __construct,     arginfo_splfixedarray_construct,ZEND_ACC_PUBLIC)
1057:   SPL_ME(SplFixedArray, __wakeup,        arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
1058:   SPL_ME(SplFixedArray, count,           arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
1059:   SPL_ME(SplFixedArray, toArray,         arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
1060:   SPL_ME(SplFixedArray, fromArray,       arginfo_fixedarray_fromArray,   ZEND_ACC_PUBLIC|ZEND_ACC_STATIC)
1061:   SPL_ME(SplFixedArray, getSize,         arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
1062:   SPL_ME(SplFixedArray, setSize,         arginfo_fixedarray_setSize,     ZEND_ACC_PUBLIC)
1063:   SPL_ME(SplFixedArray, offsetExists,    arginfo_fixedarray_offsetGet,   ZEND_ACC_PUBLIC)
1064:   SPL_ME(SplFixedArray, offsetGet,       arginfo_fixedarray_offsetGet,   ZEND_ACC_PUBLIC)
1065:   SPL_ME(SplFixedArray, offsetSet,       arginfo_fixedarray_offsetSet,   ZEND_ACC_PUBLIC)
1066:   SPL_ME(SplFixedArray, offsetUnset,     arginfo_fixedarray_offsetGet,   ZEND_ACC_PUBLIC)
1067:   SPL_ME(SplFixedArray, rewind,          arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
1068:   SPL_ME(SplFixedArray, current,         arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
1069:   SPL_ME(SplFixedArray, key,             arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
1070:   SPL_ME(SplFixedArray, next,            arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
1071:   SPL_ME(SplFixedArray, valid,           arginfo_splfixedarray_void,     ZEND_ACC_PUBLIC)
1072:   PHP_FE_END
1073: };
1074: /* }}} */
1075:
1076: /* {{{ PHP_MINIT_FUNCTION */
1077: PHP_MINIT_FUNCTION(spl_fixedarray)
1078: {
1079:   REGISTER_SPL_STD_CLASS_EX(SplFixedArray, spl_fixedarray_new, spl_funcs_SplFixedArray);
1080:   memcpy(&spl_handler_SplFixedArray, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1081:
1082:   spl_handler_SplFixedArray.offset          = XtOffsetOf(spl_fixedarray_object, std);
1083:   spl_handler_SplFixedArray.clone_obj       = spl_fixedarray_object_clone;
1084:   spl_handler_SplFixedArray.read_dimension  = spl_fixedarray_object_read_dimension;
1085:   spl_handler_SplFixedArray.write_dimension = spl_fixedarray_object_write_dimension;
1086:   spl_handler_SplFixedArray.unset_dimension = spl_fixedarray_object_unset_dimension;
1087:   spl_handler_SplFixedArray.has_dimension   = spl_fixedarray_object_has_dimension;
1088:   spl_handler_SplFixedArray.count_elements  = spl_fixedarray_object_count_elements;
1089:   spl_handler_SplFixedArray.get_properties  = spl_fixedarray_object_get_properties;
1090:   spl_handler_SplFixedArray.get_gc          = spl_fixedarray_object_get_gc;
1091:   spl_handler_SplFixedArray.dtor_obj        = zend_objects_destroy_object;
1092:   spl_handler_SplFixedArray.free_obj        = spl_fixedarray_object_free_storage;
1093:
1094:   REGISTER_SPL_IMPLEMENTS(SplFixedArray, Iterator);
1095:   REGISTER_SPL_IMPLEMENTS(SplFixedArray, ArrayAccess);
1096:   REGISTER_SPL_IMPLEMENTS(SplFixedArray, Countable);
1097:
1098:   spl_ce_SplFixedArray->get_iterator = spl_fixedarray_get_iterator;
1099:
1100:   return SUCCESS;
1101: }
1102: /* }}} */
1103:
1104:
1105: /*
1106:  * Local variables:
1107:  * tab-width: 4
1108:  * c-basic-offset: 4
1109:  * End:
1110:  * vim600: noet sw=4 ts=4 fdm=marker
1111:  * vim<600: noet sw=4 ts=4
1112:  */
```

```
/*
  +----------------------------------------------------------------------+
  | PHP Version 7                                                        |
  +----------------------------------------------------------------------+
  | Copyright (c) 1997-2018 The PHP Group                                |
  +----------------------------------------------------------------------+
  | This source file is subject to version 3.01 of the PHP license,      |
  | that is bundled with this package in the file LICENSE, and is        |
  | available through the world-wide-web at the following url:           |
  | http://www.php.net/license/3_01.txt                                  |
  | If you did not receive a copy of the PHP license and are unable to   |
  | obtain it through the world-wide-web, please send a note to          |
  | license@php.net so we can mail you a copy immediately.               |
  +----------------------------------------------------------------------+
  | Author: Antony Dovgal <tony@daylessday.org>                          |
  |         Etienne Kneuss <colder@php.net>                              |
  +----------------------------------------------------------------------+
*/

/* $Id$ */

#ifndef SPL_FIXEDARRAY_H
#define SPL_FIXEDARRAY_H

extern PHPAPI zend_class_entry *spl_ce_SplFixedArray;

PHP_MINIT_FUNCTION(spl_fixedarray);

#endif  /* SPL_FIXEDARRAY_H */

/*
 * Local variables:
 * tab-width: 4
 * c-basic-offset: 4
 * End:
 * vim600: noet sw=4 ts=4 fdm=marker
 * vim<600: noet sw=4 ts=4
 */
```

```c
/*
   +----------------------------------------------------------------------+
   | PHP Version 7                                                        |
   +----------------------------------------------------------------------+
   | Copyright (c) 1997-2018 The PHP Group                                |
   +----------------------------------------------------------------------+
   | This source file is subject to version 3.01 of the PHP license,      |
   | that is bundled with this package in the file LICENSE, and is        |
   | available through the world-wide-web at the following url:           |
   | http://www.php.net/license/3_01.txt                                  |
   | If you did not receive a copy of the PHP license and are unable to   |
   | obtain it through the world-wide-web, please send a note to          |
   | license@php.net so we can mail you a copy immediately.               |
   +----------------------------------------------------------------------+
   | Authors: Marcus Boerger <helly@php.net>                              |
   +----------------------------------------------------------------------+
 */

#ifndef PHP_SPL_H
#define PHP_SPL_H

#include "php.h"
#include <stdarg.h>

#define PHP_SPL_VERSION PHP_VERSION

extern zend_module_entry spl_module_entry;
#define phpext_spl_ptr &spl_module_entry

#ifdef PHP_WIN32
# ifdef SPL_EXPORTS
#   define SPL_API __declspec(dllexport)
# elif defined(COMPILE_DL_SPL)
#   define SPL_API __declspec(dllimport)
# else
#   define SPL_API /* nothing */
# endif
#elif defined(__GNUC__) && __GNUC__ >= 4
# define SPL_API __attribute__ ((visibility("default")))
#else
# define SPL_API
#endif

#if defined(PHP_WIN32) && !defined(COMPILE_DL_SPL)
#undef phpext_spl
#define phpext_spl NULL
#endif

PHP_MINIT_FUNCTION(spl);
PHP_MSHUTDOWN_FUNCTION(spl);
PHP_RINIT_FUNCTION(spl);
PHP_RSHUTDOWN_FUNCTION(spl);
PHP_MINFO_FUNCTION(spl);


ZEND_BEGIN_MODULE_GLOBALS(spl)
  zend_string *autoload_extensions;
  HashTable   *autoload_functions;
  intptr_t     hash_mask_handle;
  intptr_t     hash_mask_handlers;
  int          hash_mask_init;
  int          autoload_running;
ZEND_END_MODULE_GLOBALS(spl)

ZEND_EXTERN_MODULE_GLOBALS(spl)
#define SPL_G(v) ZEND_MODULE_GLOBALS_ACCESSOR(spl, v)

PHP_FUNCTION(spl_classes);
PHP_FUNCTION(class_parents);
PHP_FUNCTION(class_implements);
PHP_FUNCTION(class_uses);

PHPAPI zend_string *php_spl_object_hash(zval *obj);

#endif /* PHP_SPL_H */

/*
 * Local Variables:
 * c-basic-offset: 4
 * tab-width: 4
 * End:
 * vim600: fdm=marker
 * vim: noet sw=4 ts=4
 */
```